Broadview
www.broadview.com.cn

Geekbang
极客邦科技
InfoQ

聊聊“架构”

ARCHITECTURE TALKS

王概凯 / 著

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

聊聊架构

ARCHITECTURE TALKS

王概凯/著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

架构是如何运作并影响人们的日常生活的，在软件行业中架构是如何运作的？架构又是如何指导代码编写的，如何把架构应用在软件工程实践上？带着这些疑问，本书通过大量的实例一步一步揭示出架构背后的原理，以及架构在软件行业的发展，并通过企业实例来展示软件架构的实际应用。本书没有高深的词汇，不仅适合 IT 从业人员阅读，也适合其他行业的人士阅读。尤其对于想从事架构工作的人而言，是一本不可多得的参考材料。

图书在版编目（CIP）数据

聊聊架构 / 王概凯著. —北京：电子工业出版社，2017.4
ISBN 978-7-121-31122-2

Ⅰ. ①聊… Ⅱ. ①王… Ⅲ. ①软件开发—普及读物 Ⅳ. ①TP311.52-49

中国版本图书馆 CIP 数据核字(2017)第 057527 号

策划编辑：张春雨
责任编辑：徐津平
印　　刷：中国电影出版社印刷厂
装　　订：中国电影出版社印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：720×1000　1/16　印张：15.5　字数：256.2 千字
版　　次：2017 年 4 月第 1 版
印　　次：2017 年 5 月第 2 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

谨以此书献给我的父亲和母亲！感谢两位老人的照顾和爱护，本书才得以顺利完成。

本书也献给我内人！作为本书的第一位读者，她提了很多建议，并做了很多文字校对工作，本书有她的一份功劳。没有她的体贴和关心，以及支持和帮助，本书也无法顺利完成。

本书也献给我可爱的小朋友们！在爸爸写累了的时候，有你们陪爸爸玩耍放松，你们总是能给家庭带来无尽的快乐。愿小朋友们平安、快乐、健康地长大！

序

在软件行业，架构师和软件工程师是非常辛苦的职业。一方面新技术层出不穷，另一方面业务需求也层出不穷，让人疲于应付。导致的后果就是常常加班，生活质量低下。只有曾经身在其中的人，才能够体会其中的酸甜苦辣。

在软件行业经历过这么多年，也看到了软件行业普遍存在的一些问题，总觉得自己应该为这个行业贡献一点点力量。不期望能够改变这个行业，能够引起一点点思考也是好的，如果能够帮助一些软件从业者提升工作和生活质量，就超出期望值了。

把自己的想法写出来的过程是痛苦的，从来没有写文字的习惯，也没打算过写书，因此愈发艰难。年初时基于以上同样的想法，在 InfoQ 投稿写了《架构漫谈》专栏，和大家分享一下自己对软件架构的思考，以为算是交差了。不料 InfoQ 的郭蕾多次和我约稿，希望我能够把架构漫谈扩展成一本书。拒绝了很多次，但是脸皮实在是薄，禁不住郭蕾三番五次的游说，狠狠心答应了下来。

把文字写下来传播出去，是要承担很大责任的，一旦说得不对，伤害的就是一大片人。不愿写东西的原因大部分在此。但是想想人非圣贤，总有犯错的时候，把自己的错误暴露给大家，也是帮助大家学习。话虽如此，还是郑重声明，本书的内容都是个人的思考和个人的观点，并非学术的结论，请各位读者不要当作结论全盘接受。反而读者应该质疑书中的各种观点，尽量自行思考，如此才会有所收获。本书的目的也仅仅是为了引发大家的思考。

思及自身水平有限，文字功底也差，难免伤人慧命，深感惭愧和惶恐！望各位读者，鉴其愚诚，不吝慈悲指正！

王概凯　Kevin

前言

现代的软件从业者，都受过良好的计算机和软件方面的教育。但是现代的计算机和软件方面的教育，基本上都是从科学研究领域脱胎出来的，教育的目的也理所当然的主要是为科学研究领域服务。而随着社会的发展，软件不断地渗透到不同的业务领域，涉及普通人生活的方方面面。以科学研究为目的的软件教育，和日益深入人们生活的软件应用，产生了很大的隔阂。以致很多计算机和软件专业毕业的学生，进入企业工作后，总是感叹学校所学习的知识派不上用场，必须得重新学起，才能够达到企业的要求。

而这些重新学习的内容，又往往是以技术为主的。技术的更新换代太快，往往也导致想跟上新技术的我们力不从心。技术和业务的关系是怎样的？业务又是怎么运作的？很少有人去问这些问题。即使有人问了，也很难有人可以提供建议。

软件技术学习到一定的地步，又会发现软件架构是一个门槛。一直以来，在软件行业，对于什么是架构有很多的争论，每个人都有自己的理解。甚至很多架构师一说架构，就开始谈论应用架构、硬件架构、数据架构等。而事实上，架构在软件发明前就早已存在了。众说纷纭，莫衷一是，这也给大家带来了很多困扰。

业务和架构，是压在软件从业人员身上的两座大山。而软件从业人员手上却只有一个武器：技术。可是这个武器还时灵时不灵，就好像金庸小说《天龙八部》中段誉的六脉神剑，并不总是能够解决问题，有时还会带来麻烦。

软件并不是虚无缥缈的东西，它和现实生活是紧密相关的。业务和架构都是处理人的问题。而技术人员最讨厌处理的就是人的问题，心里面厌恶，却又无法逃避。因为这个排斥的心理，工作中始终想避开和人有关系的地方。因此在做技术之前，还需要做一些准备工作，用来连接现实生活，联系上人，让大家知道处理人的问题并不可怕。建立了这个相关性，每个人就都可以自行思考了。

不仅人类受限于自身的生命周期，凡事都有其生命周期。理解了生命周期，

就可以看到很多隐藏在背后的规律，以及这些规律之间的联系。因此，本书试图从生命周期入手，描绘出一张整体的画卷，帮助包括技术人员在内的读者定位自己处于什么地方，自己在起什么作用，别人又在什么地方，他们又在起什么作用。明白了自己的位置和别人的位置，自然也就清楚自己有什么，缺什么，要往哪个地方走，从哪些地方入手了。所谓“知己知彼，百战百胜”，知道这些后，与人打交道时也就有了自己的思考方式，能够进行独立思考，对业务也不再厌恶以致逃避，而是为能帮助业务人员分析及解决问题而自豪。

本书虽然不是技术书籍，不谈技术，却是以帮助技术人员为出发点的。本书的内容可以作为连接技术人员和现实世界的桥梁，使技术人员不再悬在空中使不出力。对于非技术人员，本书可以帮助其理解软件开发的特殊性，拉近与技术人员的距离，能够更有针对性地与技术人员合作。

当然，读完本书不会使读者突然学会神功，打通任督二脉。因为每个人的成长，最终还是要靠自己的思考和实践。本书的思考也不能够代替读者自己的思考，在解决某个业务问题时也无法从书中直接找到答案。本书可以提供给读者的是一个思考的出发点，一个思考的方向，一个思考的角度，使得读者不再惧怕或排斥业务，并可以像业务人员一样思考，和架构师一样思考，不再受困于业务和架构，甚至是技术本身。如果本书能够帮助读者跨过这个门槛，并从这里开始展开思考，那么本书的目的就达到了。

目录

第一部分

认识架构

树立对架构的正确理解，建立对架构的基本认识

第 1 章　生命周期

这个世界是如此多彩，每一个个体都是如此美妙，打动人心。这个世界每时每刻都在不断地变化，正因为变，我们每时每刻对这个世界都有新的感受，才觉得她是这么的美丽。

每个个体的变化都是不同的，都有自己的特点。可是所有事物的变化都有一个共同的特点：从出生开始，到消亡结束。用佛家的话说，也就是“生住异灭”。所有的事物一旦出现了，就必然会消亡。而正因为有不断的消亡，才会有不断的出生，所谓“生生不已”。这一切都是在人们掌控之外的，正如人们无法决定自身的生灭。因为人们无法去掌控她，她才会如此美丽，打动人心。

一个事物一旦出生，就必然会长大，变异，一旦长大，就面临着衰老，接下来就是消亡了。这个过程就称为一个事物的生命周期。生命周期（Life Cycle）实际上是近代出现的概念，从西方传入。自古以来，我们称之为生灭。为了叙述上的方便，后续所指的生命周期，实际上指的就是生灭。

人类自古以来就在研究自身的生灭问题，也就是生死问题。这个问题隐藏在每个人的言行举止中，人们经常为之深深地恐惧。这种恐惧的心理驱使着人们做出种种的努力，来延缓甚至避免这个结果，虽然人们明知无法避免。

人类为了保持活着的状态，因而需要满足身体生存的必要条件，那就是要获取能量，并能在复杂的自然条件下保存身体，并生存下来。这个原始的动力，从没有离开过我们。我们必须时刻清楚这一点。不了解这一点，人们很多时候就无法理解自己的行为，因为这个问题深深地隐藏在人们的潜意识当中。看看我们每天的活动，起床，穿衣，吃饭，工作，睡觉，无一不是为了维持身体生存而做的动作。

等一等……人们每一天的活动，是不是也是一个生命周期呢？确实是的，生命周期无处不在。从嚼一口饭，到捡起一支笔，到眨一下眼睛，都是一个生灭，

都是一个生命周期。我们一天里所有的子生命周期组成了这一天，一辈子三万六千天的大生命周期，由三万六千个每天的子生命周期组成。而每一个生命周期的结束，往往又会导致其他事物的生命周期开始。比如人们吃一口饭，饭消失了，产生了能量，人们通过活动，消耗掉能量的同时又产生了新的事物。

所以生命周期并不是孤立的，生命周期里包含生命周期，一个生命周期消亡会产生另一个生命周期。我们生存在一个生灭变化的世界，需要认真认识生命中的这些生命周期，以及生命周期之间的关系，并保持敏锐的观察。这会让我们更加清晰地理解这个世界，并能够更和谐地和这个世界相处。

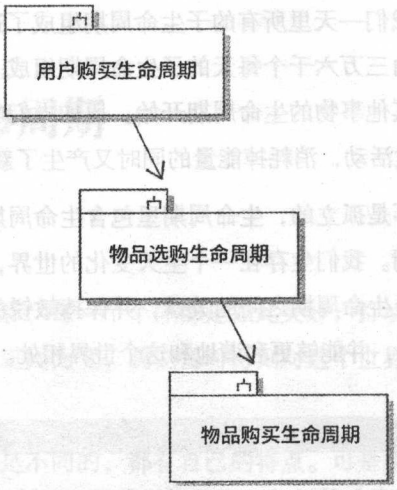
1.1　生命周期的识别

生命周期里面包含有各种活动，这些活动是推进这个生命周期的必要因素，并且这些活动之间有时间上的推进关系。比如，用户购物这个场景，从用户进入到商店，进行浏览、询问、购买等活动，到离开商店，都是按时间顺序一步一步发生的。每个发生的活动，本身也是一个生命周期。

生命周期总是有一个主体，因此更多的是明确指出某某的生命周期。所以识别出生命周期的主体是至关重要的事情。如上文提到的购物场景，从用户进入到商店，进行浏览、询问、购买等活动，到离开商店，这个生命周期的主体是谁？这个生命周期的主体并非用户本身，而是用户的一次购买活动。如果是用户的生命周期，则变成了用户的生成和消亡了。

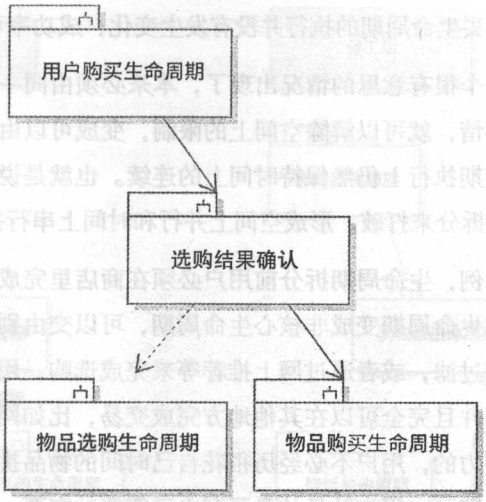
1.2　核心与非核心生命周期

一个生命周期里面的活动可以进行拆分，拆分的原则就是形成若干个新的生命周期，每个新的生命周期都有自己的主体。比如，再承上例，用户的购买活动又可以拆分为<物品的选购>和 <物品的购买>两个小的生命周期，分为两个阶段。拆分过程如下图所示，箭头表示执行的顺序。



一旦拆分为小的生命周期我们就会发现，其中一个生命周期的主体也发生了改变：<物品的选购>这个生命周期的主体为用户的物品意向，以用户进店为开始，以意向确定或放弃为终；<物品的购买>这个生命周期的主体为用户的购买活动，主体并没有发生改变，但是主体的生命周期活动内容减少了，变成以物品确定为开始，以交易成功或失败为终。这两个小的生命周期以物品确定这个行为分别为终始，组成了一个大的生命周期：用户的购买活动。

我们也把拆分之后主体不变的子生命周期，称为核心生命周期。主体改变的子生命周期，称之为非核心生命周期。寻找核心生命周期的过程，实际上也就是一个发现内聚的过程。对于非核心生命周期，本身内部也需要寻找出它自己的核心生命周期，这同样也是发现内聚的过程。如下图所示，虚线表示非核心生命周期，实线表示核心生命周期，箭头表示执行的顺序，树的遍历的次序从左至右。



拆分出来的每个生命周期各自都有自己的边界，不会影响到其他的生命周期，因为各自的变化都在自己的生命周期内确定。每个主体的生命周期活动的变化都累积在该主体本身，这就是所谓的内聚。也就是说，要做到内聚，必然要先确定生命周期的主体和生命周期本身。确定了这两样，内聚就是自然而然的结果。所以，当讨论内聚的时候，实际上就是讨论生命周期主体以及该生命周期内所有的活动，我们一定要意识到这一点。

1.3　生命周期与分工

在把一个大的生命周期拆分为多个小的生命周期后，核心生命周期活动的执行都严格地在时间上连续。而非核心生命周期的管理，则围绕着核心生命周期形成了一个树状结构。随着大的生命周期的拆分，树在逐渐地长大。

拆分之前，生命周期内部的活动都是由该生命周期的主体来执行的。在拆分之后，核心生命周期还是由原生命周期的主体来执行，但是非核心生命周期的主体发生了变化。也就是说生命周期的运营和执行产生了分离。

如上例，物品选购结果的确定是核心生命周期需要的，也就是说执行还是必须在购买之前。但是选购结果确定之前的种种行为，如逛街、进店挑选等，也就是选购生命周期的运营，则完全可以交由别人来执行。现今的网上购物、智能推荐等都是在这个地方发力。这样选购的运营就可以和购买生命周期的执行在空间

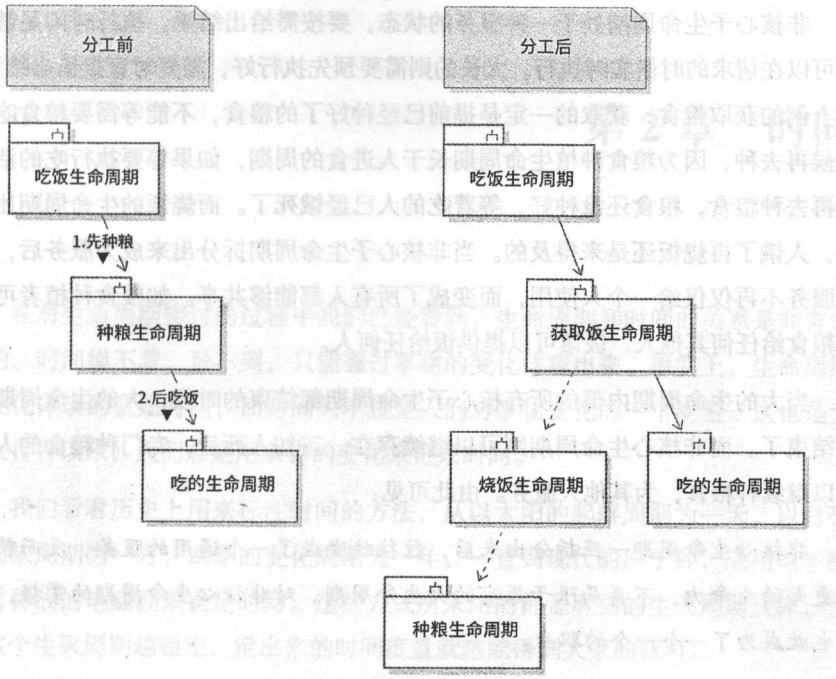
和时间上并行，购买生命周期的执行并没有发生变化，成功率可能会更高。

拆分之后，一个很有意思的情况出现了，本来必须由同一个主体在时间和空间上连续执行的事情，就可以解除空间上的限制，变成可以由不同主体来执行，但是原主体生命周期执行上仍然保持时间上的连续。也就是说空间上连续的限制可以通过生命周期拆分来打破，形成空间上并行和时间上串行执行的状况。

比如还是承上例，生命周期拆分前用户必须在商店里完成购买。一旦生命周期分拆之后，选购生命周期变成非核心生命周期，可以交由别人执行，由别人代替用户上街选择和过滤，或者通过网上推荐等来完成选购，用户只需要最后确定物品的挑选即可，并且完全可以在其他地方完成交易，比如网上。商品的选购往往是最花时间和精力的，用户不必经历很花自己时间的物品挑选过程，只需要给出想要的物品型号即可。用户的目的是买到自己需要的东西，而不是选购本身。核心生命周期变得更加简短了。

拆分同时会带动产生很多新的有意思的生命周期，形成新的技术，比如商品的规范化、货物的配送等。有的时候则反过来，新的技术出现导致拆分变成可能。比如照相机的出现，使得人们眼睛看到物体的生命周期发生了根本性的分拆；车的出现，使得人们从一地到另一地的生命周期发生了拆分，不再全靠步行中每一步的生命周期累加；软件的出现，互联网的出现，使得人们生活方方面面的各种生命周期都发生了分拆……

再比如，吃饭是人的核心生命周期，必须要自己执行，别人无法代替。为了完成吃饭这个生命周期活动，在没有产生社会分工之前，每个人自行种粮，用自己种的粮食做饭吃，这个过程按照时间顺序连续发生。我们把这个过程拆分之后，可以发现里面只有“吃”是人类核心的生命周期活动，别人无法代替，必须自己执行，而其他活动都是为吃服务的。随着社会分工的发生，拆分出来种粮的生命周期管理、烧饭的生命周期管理等，可以交给其他人来执行。人们可以通过购买粮食和请人烧饭来直接获取执行的结果，以推进自己吃饭的核心生命周期活动。这就产生了管理和执行的分离。如下图所示，虚线代表非核心生命周期，实线代表核心生命周期，箭头代表执行顺序，树的遍历为从左至右。



非核心生命周期的管理和执行、核心生命周期的管理和执行可以在时间和空间上并行，但执行仍然还是要依照拆分之前的大生命周期的顺序，没有发生改变。比如粮食种植这个非核心生命周期分工给别人了，但是烧饭生命周期执行前先要有粮食，因为“巧妇难为无米之炊”，还是得等别人的粮食种出来之后才能够购买获得，而饭则要等别人烧好了才能吃。

从生命周期内部来看，生命周期的每个阶段是随着时间的推进而发生变化的，因此呈现出来的是一个按时间顺序发展的状况。比如人的吃饭、睡觉等活动，虽然都是独立的活动，但是每天早、中、晚三餐饭都是按顺序发生的。每天晚上也都需要睡觉来补充体力。不可能边吃饭边睡觉，或者三餐一起吃。这些活动需要按照人的生命周期规律逐步地按顺序发生。

反过来，当启动一个核心子生命周期的时候，必须把树上该节点的所有父生命周期启动才可以，也就是所谓的业务流程，因为大生命周期本身内部活动还是按照时间顺序连续发生的。比如，用户下订单之前，必须先要给出选定的商品列表；吃饭之前，先要有饭。

非核心子生命周期处于一种服务的状态，要按需给出结果。执行时间足够短的可以在请求的时候实时执行，太长的则需要预先执行好，需要时直接给出结果。如上例的获取粮食，获取的一定是提前已经种好了的粮食，不能等需要粮食吃的时候再去种，因为粮食种植生命周期长于人进食的周期，如果等要执行吃的活动时再去种粮食，粮食还没种好，等着吃的人已经饿死了。而烧饭的生命周期比较短，人饿了再烧饭还是来得及的。当非核心子生命周期拆分出来成为服务后，这个服务不再仅仅给一个人使用，而变成了所有人都能够共享。如粮食种植者可以卖粮食给任何其他人，饭馆可以提供饭给任何人。

当大的生命周期内部的所有核心子生命周期都结束的时候，大的生命周期也就结束了。而非核心生命周期则可以继续存在。正如人死后，专门种粮食的人还可以继续种粮食，为其他人服务。由此可见：

非核心生命周期一旦拆分出来后，往往就变成了一个通用的服务，反而获得了更大的生命力，不再局限于原有的大生命周期。对非核心生命周期的掌握，慢慢也就成为了一个一个的职业。

而原有的大生命周期则变得更加精简，可以更加专注于自己的核心生命周期活动，以节省更多的时间。

第2章　时间

在对生命周期探讨的过程中我们已经看到，生命周期和时间的关系是非常紧密的。时间摸不着，看不到，只能通过事物的变化体现出来。事实上，生命周期的变化体现的就是时间，而时间只不过是人们对事物变化的一个度量。这也是为什么自古以来，人们总是用事物的变化来记录时间。

我们看看历史上用来标注时间的方法，从以太阳的起落周期为一天，以月亮的圆缺周期为一月，四季的变化周期为一年，一直到现代的原子钟，采用电子跃迁时释放的电磁波来确定时间。这些方式所采用的都是事物的生灭周期规律。并且这个生灭周期越稳定，定出来的时间度量就越能得到大家的认可。

所以人们对生死的恐惧，就转而形成了对时间的恐惧。在人的短短一生中，维持生命必需的活动占据了绝大部分的时间，那我们怎样在剩下的时间里做出一些努力呢？无论承认与否，这是每个人内在的焦虑。为减轻这个焦虑而做出的思考和努力，就变成了人类进步的动力。

很多科学研究都在探寻时间，时间本身究竟是怎么回事？我们是否可以回到过去，甚至去到遥远的未来？为了延续人类，很多科学家积极地探索外太空，因为地球终究也会消失的，人类要有其他的地方可以居住。

一个人静静地坐在池塘一角，看着平静的池塘。随着鱼儿的活动，水面时而浮起涟漪，又转瞬而逝。天空，云影徘徊，变动不居。微风轻吹，直吹到心灵的深处。每个事物，都遵循着自己的规律，遵循自己的生命周期，该来的来，该走的走。此时眼泪不禁从双眼滑落：世界正是因为不断地来来去去，生生灭灭，才是这么多彩，才是这么吸引人。

每个人也都在努力与时间赛跑，尽可能地在事物消逝前多占有一会儿，但是不管人们愿不愿意，事物最终都会离我们而去。过去的已过去，未来的还没来，只有当下的一刻是最真实的，大自然是我们学习的榜样。

人类的生命很短，百年也只有短短的三万六千天。自古以来人们寻求长生不老，希望能够延长自己的生命，得到的结果总是失望。其实反过来思考，如果人类活得足够长，会有这么大的动力去创造出这么多的新技术吗？反而寿命越短，越是焦虑，越希望能够得到更多的东西，物质技术也越发达，分工也越严密细致。得到越多，越希望能够留住，越留不住就越焦虑，形成了一个很糟糕的循环。物质的发达对人来说似乎并不一定是好事。

大部分人都不愿意接受一切都终将消逝的事实，总想活得更久，占有更多，享受更多。在人们短短的一生中，如何延长自身的生命呢？一个办法就是尽可能做出更多的成就，能够让更多的人生活得更好。在同样的时间内创造出更多的产出，相当于把自己的生命延长了。另一个办法就是创造下一代，这在某种程度上也可以视为延长了时间，也是人类延续的动力。这个动力，也推动了人类生命周期的拆分与人类架构的发展。

尽可能做出更多的成就的办法，就是尽量做自己最擅长的事情，以期获得最大的产出，人类的社会化正由此而来。

第 3 章　为什么会产生架构

延长每个人自己生命的时间，就要尽可能做自己擅长的事情，以及产生下一代。

想象一下，人类在最早期，每个人都完全独立生活，衣、食、住、行等等全部都自己搞定，整个人类都是独立的个体，不相往来。每个人都局限在自己所在的地域，独立完成自己的生活必需品的生命周期以支持自身的核心生命周期。每个人为维持生命所做的活动很多，而由于人类结构的原因，一个人同时只能够做一件事情，这会导致很多时间上可以并行的工作变成了串行。

正如我们前面生命周期中所讨论的，除了那些别人无法替代必须要由自己来完成的核心生命周期活动（如吃饭、睡觉等）外，把其他的生命周期活动并行起来，在同样多的时间内做更多的事情，也相当于延长了自己的生命。如何做到这一点呢？

3.1　分工

产生下一代的要求就会导致男女群居，这个时候自然就出现了性别分工，男性和女性所做的事情就会有一定的区别。比如男性体力好、强壮，就会更多地在野外获取食物。女性细致、耐心，更多地做一些衣物缝制、食物制作、照顾下一代等工作。还会出现地域的分工，比如有些地方适合种粮食，有些地方适合种棉花等。因此，就从原来每个人都要自行获取所有的生活必须品，变成了多个人可以在时间和空间上并行地进行。参与分工的人就可以专心地做好自己擅长的事情，或者所处的不同地域所擅长的事情。

可是人们每天生活的基本需求没有发生变化，还是衣食住行等生活必须品。一旦多人分工配合作为生存的整体，力量就显得强大多了，所以也自然地形成了族群：有些人种田厉害，有些人制作工具厉害，有些地方适合产出粮食，有些地方适合产出棉花等，就自然形成了人的分群，地域的分群。当分工发生后，实际上每个人的生产力都得到了提高，因为做的都是每个人擅长的事情，相应的每个

人的生命都得到了延长。整个人群的生产力和抵抗环境变化冲击的能力都得到了增强，某种程度上人的寿命也同时会得到增长。

既然分工发生了，原来由一个人执行生存所必需的所有的事情，就变成了很多不同分工的角色合作完成这些事情，人的生命周期活动发生了拆分。拆分后每个人只需要执行必须自己做的事情和自己擅长的事情。这些人必须要通过某些机制整合在一起，让每个人都得以完成生存所必需的事情，这实际上也导致了交易的发生。交易这部分暂时就不在这里展开了，后续再讨论。

当每个人都必须自己完成所有生活必须的活动的时候，是没有架构的。一旦产生了分工，就把原来每个人所必须做的人类非核心生命周期工作，切分成由不同角色的人分别来完成，最后再通过交易，使得每个个体都拥有生活必须品，而不需要每个个体都做所有的事情，只需要每个个体做好自己擅长的事情，并具备一定的交易能力即可。这样每个人为自己核心生命周期所花的时间就少多了，有更多的时间用来提升核心生命周期的质量，从而使生命得到了某种程度的延长。这实际上就形成了人类社会的架构。

从以上的例子可以看出来，人有能力按照自己的意愿控制其他事物的生命周期，并重新组合，以在某种程度上延长自己的生命周期，这一点是超越其他生命的，也是人类架构产生的必要条件。

3.2 分工和生命周期

再从生命周期上来看，原来需要一个人完成所有的事情，生命周期被细分为很多小的非核心生命周期。这些小的非核心生命周期，就可以由另外一个人来负责，在空间上和时间上都得以并行，原来的那个人只需要获得另一个人的结果就可以了。比如一个人必须要先种田，完成粮食的产生，并消费粮食，结束粮食的生命周期才能完成自己的能量获取，用以维持生命。此时粮食的生命周期和人的生命周期都是由一个人来管理，本来可以并行的事情，由于人本身的限制，也可以说由于空间的限制，变成了串行。当生命周期切分后，粮食生产的生命周期就分解为粮食的产生生命周期和粮食的消费生命周期。人就不必只依赖于自己生产的粮食，大大拓宽了获取的渠道，而人本身的生命周期并没有受到影响，可是却大大地节省了时间，延长了自己的生命。

第 4 章　什么是架构

由上文可见，推动人类架构产生的动力在本质上还是人类对时间的恐惧。而人类为克服这个恐惧，尽可能地去延长自己的时间，去提升自己的生命质量，毫无疑问就变成了人类内在的动力。

4.1　架构产生的条件

细分一下架构产生的动力，基本有以下几点：

（1）必须由人执行的工作。

不需要人介入，就意味着不需要改造；不需要节省时间，也就不需要另外再产生架构了。大自然会自行调整自然界本身的架构。

（2）每个人的时间有限。

每个人都有自己的强项，个人的产出受限于最短板，并且由于人的结构限制，同时只能专注于做好一件事情。比如，人类虽然有两只眼睛，但是只能同时专注于一件事物；两只手无法同时做不同的事情，虽然有少部分人可以左手画圆右手画框，但这不是普遍现象。人们为了减少自身生命周期活动的时间投入，必然会把自己不擅长的非核心生命周期工作分解出去，给擅长于这些工作的角色来完成，从而缩短自己所花的时间。

（3）对目标系统有更高的要求。

如果满足于现状，没有时间压力，也就不需要进行架构了。

（4）目标系统的复杂性使得单个人完成这个系统时会受限于时间。

如果自己就可以完成系统的提高，不需要其他人参与，也就不需要架构的设计，这样的人只能算是工匠。因为并不一定会产生分工的思考和尝试，并且一般这个工作对时间的要求也不迫切。当足够熟练之后，这样的人也会有一定的架构

分工思考，但考虑更多的是如何提高质量，提高个人时间的效率。

有人可能会说，一个人建一栋房子，对目标系统进行分解，自己采购材料，自己搭建，难道也不算架构嘛？架构的思考来源于对生命周期的识别，以及对生命周期的拆分。如果没有生命周期拆分的思考，就不能算架构。世界上第一个建房子的人一定是有架构思考的，因为要建房子这件事本身就是一个架构思考，这是对地球上空间的主动切分，对人自身生命周期活动所占空间的切分。但如果只是出于对别人的模仿，则并不会有架构思考。如果对时间不敏感的话，比如已经有住的地方，有足够的时间慢慢造，也并不一定会导致架构思考的产生。如果有足够的自觉，而且足够熟练的话，确实会产生架构的思考，因为这样对于提高生产力是有帮助的，可以缩短建造的时间，并会提高房子的质量。事实上建筑的架构就是在长期进行这些活动后，积累下来的实践。思考的结果，往往会导致新技术的产生，或者新技术的应用。

当上述四个条件同时成立，一定会产生人类的架构。从这个层面上来说，人类的架构是人类发展过程中，由懵懵懂懂、被动地去认识这个世界，变成主动地去认识，并以更高的效率去改造这个世界的方法。以下我们再以建筑为例加强一下理解。

最开始人类是住在山洞里或住在树上的，主要是为了躲避其他猛兽的攻击，以及减少自然环境的变化对人类生存的挑战。山洞和树是大自然所形成的自然结构。为了解决人类生存的问题，人类开始模仿大自然，在平地上用树木和树叶来建立隔离空间的设施，这就是建筑的开始。但是完全隔离也不好，毕竟人类的生命周期完成，离不开自然界。完全隔离会导致进出不方便，也不便于观察周围，慢慢就产生了门窗等设施，形成了新的技术，新的生命周期拆分。

建筑的本质就是从自然环境中划出一块独占的空间，但是仍然能够通过门窗等和自然环境保持沟通。这个时候建筑架构就已经开始了，建筑的技术也跟着发展。

建筑架构按照人类生命周期的需求，对地球上的空间进行切分，并通过门窗、地基等技术，保持和地球以及空间的有机的沟通。

人类的必要生命周期活动，如吃、睡等，是和时间相关的，自然也就会根据人的生命周期规律按照时间来划分内部空间。

比如当人类开始学会用火之后，茅棚里自然而然慢慢就会被进一步切分，一部分用来烧饭，一部分用来生活。当人的排泄慢慢移到室内后，洗手间也就慢慢地出现了。这就是建筑内部的空间切分。这个时候人们对建筑的需求也就逐渐越来越多，空间的切分也会变成很多种，组合的方式也会有很多种，比如个人居住的房子，群居所产生的宗教性质的房子，集体活动的房子等等。这个时候人们开始有意识地去设计房子，建筑架构师就慢慢地出现了。一切都是为了满足人类越来越高的需求，提升质量，减少时间浪费，更有效率地切分空间，并且让空间之间更加有机地进行沟通，让人类可以更好地享受生命，享受时间，完成核心生命周期活动。这就是建筑的架构以及建筑的架构的演变。

4.2　什么是架构

总结一下，什么是人类的架构呢？主要包含以下几个要点：

（1）根据要解决的人类的问题，对目标系统的边界进行界定。问题主体的确定一般就能够确定边界，问题的确定可以明确核心生命周期。

（2）围绕目标系统核心生命周期进行切分。切分的原则是要让非核心生命周期独立出来，便于不同的角色并行地开展工作。首先要在空间上进行拆分，使得空间上的管理可以并行，而时间执行上串行。因为只有空间上并行才能减少每个人的执行时间。达到空间上的并行后，进一步可以通过预先执行好结果，达到时间上并行的效果，缩短整个生命周期的执行时间。

（3）对这些切分出来的部分，确立各自的生命周期及其主体，以及负责的角色。不同角色对所负责的生命周期负全责，并具备享受该生命周期推动带来的利益，达到权责对等。每个切分出来的生命周期都应该是完整的，活动的结果都累积在该生命周期的主体上，也就是内聚。

（4）在这些拆分出来的非核心生命周期和核心生命周期之间设立沟通机制，使得这些非核心生命周期能够围绕核心生命周期，通过树状架构组装起来成为一个整体，共同为核心生命周期做出贡献。为什么是树状架构呢？因为核心生命周期本身是按照时间线性推进的，拆分后非核心生命周期会围绕核心生命周期形成树状的架构。

人类正是因为有了分工，才形成了人类社会。人类社会有自己的核心生命周

期，不同分工的人围绕在这个核心生命周期下形成了一个树状结构。人类社会因此形成了一个新的生命，并且比单个人的生命长得多。因为人类社会的生命并不因为某个人的死亡而消亡。人类生命同样也因此依赖于人类社会而得到了增长，达到了双赢。

是不是只有人类才有架构呢？大家应该也注意到了，前文的架构都加上了“人类”做了限定。自然界也是存在架构的。各种生命都有自己的生命周期，生命周期之间形成一个个的食物链，循环往复。这也是为什么叫作“周期”。这个世界就是通过一个一个事物的生命周期的推进，不断地向前发展，而不是往后看的。

人不过是自然界中的一种生命而已，在古代被称为“倮虫之长”。[1]各种生物组成了整个地球，地球则形成了一个生命体，有自己的生命周期。不同的星球又形成一个个的星系，进而组成整个宇宙。这就是大自然的架构。老子云：“人法地，地法天，天法道，道法自然”，这里的自然指的是自然如是，本来如此的意思。（现代语言中的“自然”，是从日本舶来的，是对英文“nature”的翻译，不是古代的本意。）任何事物都按照自己本身的生命周期规律向前发展。做架构也应该效法地、天、道、自然。大自然的架构是人类学习的榜样，人类设计的架构也必须遵循大自然的规律才能稳定、长久，因为人类是生活在大自然的架构之下的。违背了这一点，人类就要付出巨大的代价。

大自然的架构又是如何推进的呢？靠的就是生存在其中的每种生命，自生自灭，遵从每种生命自身的规律。而每种生命的活动，在得到自身生命周期活动利益的同时也要承担自身生命周期活动的责任，支撑生态链，权责对等。因此大自然才生机勃勃。

大自然的架构经过亿万年的推进，有其自身的拆分，而人类自身的社会架构拆分和分工还在继续。理解架构拆分背后的规律，对人类有意识地去架构是非常有帮助的，为我们指引了方向。为了区别大自然的架构，也为了行文方便，后续我们讨论的架构都是以人类架构为前提，不再特指。

这些思考同样可以在其他的行业展开，比如企业架构、国家架构、组织架构、

1 《内经》将虫分为五类，即毛虫、羽虫、倮（luó）虫、介虫、鳞虫，其中倮虫属土，人为倮虫之长。

音乐架构、色彩架构和软件架构等等。套用三国演义的一句话，合久必分，分久必合。人类架构实际上就是指人们根据自己对世界的认识，为解决某个问题，主动地、有目的地去识别问题，并根据核心生命周期进行分解、合并，以解决这个问题的实践活动。人类架构的产出物，自然就是对核心问题的提炼和分析，对核心生命周期的识别，以及解决问题的方案。它包括拆分的原则以及理由，沟通合并的原则和理由，拆分出来的各个部分的生命周期确立，以及所对应的推动角色和角色所需要的核心能力，还有对这些拆分的落地等。

4.3　架构的生命周期

从架构本身来看，架构也是人类活动的一种分工，人类架构的体现就是组织架构。因此，架构本身也有其自身的生命周期，有其自身的规律。人类的架构总是在人类的业务遇到瓶颈的过程中产生，在瓶颈的解决中应用，在业务消亡时，它也跟着消亡。架构是对业务生命周期的拆分，自然业务的生命周期就是架构的生命周期。每种业务，它的架构拆分是确定的，区别只在于规模的不同，树的高度不一样。这就是为什么同行业的业务，往往都是一样的架构。有了这一特征，在某个行业开始一个业务，该行业的架构就可以直接拿过来参照实现，这就是架构的模式。

架构的生命周期可以被拆分为两个子生命周期：架构设计生命周期、架构实施生命周期。其中，架构设计生命周期是为架构实施生命周期服务的，因此架构实施生命周期是架构的核心生命周期，架构设计生命周期是非核心生命周期。非核心生命周期的并行管理，导致诞生了很多专门做架构设计的公司，专门给人做设计外包，或者做架构咨询（Consulting）。但架构的核心是实施，很多雇主为了防止自己购买的设计方案无法落实，往往会要求设计提供者同时提供架构的落地服务。吃过苦头才知道，架构只有方案是没有意义的，最重要的是把架构落地执行。

架构的设计生命周期，主要工作就是研究业务本身的生命周期。然后再根据业务面对的问题，发现瓶颈，进行架构的拆分。拆分的原则则是把非核心生命周期拆分出来，由不同的角色来负责，让人们可以并行工作。每个角色达到权责的对等，并形成不同角色各自的激励机制。架构的目的并非产生一个业务之外新的东西出来，只是为了让业务长得更加高大强壮，服务于更多的人。

架构实施生命周期，则是为了把架构的拆分落实到组织架构上，让每一个人能够按照架构的职责拆分，并行工作，执行各自的生命周期。这些角色按照树状架构协调互相之间的沟通，使得每个拆分出来的非核心生命周期的增长，都贡献给核心生命周期，同时自己也能够受益。也只有树状架构能够做到这一点。

在架构落地之后，架构就具备了生命。人们在这个架构下有条不紊地协作，共同完成业务的核心目标。

第 5 章　架构和树

树是自然界中非常常见的一种结构。看看地球上长出来的东西，基本上都是树状的，地球上生长出来的万物基本都是地球的枝叶。人体是由一个细胞分裂出来的，也是一棵树。再看看算法，基本上效率最均衡的也是树。生命基本上都和树有关系。

我们来观察一棵树，树有主干，有枝叶。主干就相当于是核心生命周期，决定了树的生死。枝叶是非核心生命周期，负责为主干获取太阳的能量。到了冬天，阳光不够、天气寒冷了，树叶就开始枯黄掉落，为树保存能量，使得树能够更好的过冬；到了春天，树叶生长，通过吸收阳光，自身长大，也为树干供应营养。同样树根也是一个树状的结构，负责从地球获取营养，自身也往下生长。我们做架构，就要从人、地、天、道这里来进行学习。

5.1　树与增长

树状结构有什么好处呢？树状结构特别适合于增长。如果是按照直线往上增长，长得越长，越容易失控，沟通成本越高，风一吹就折。如果按照水平增长，则很难获取阳光，容易被周围高的物体挡住太阳。而树状的增长，在成本方面最低，沟通的路径也没有显著的增长，带来的效果最好。树的结构也保证了分支的能量汇集到树干，保障了整棵树的生长。而架构恰恰是为了应对增长的，自然而然架构都是树状的。

从这个角度我们也可以看出，围绕着核心生命周期，把非核心生命周期剥离，也可以使得核心生命周期更加精简、轻量化，更容易应对周围环境的变化，树干也更容易长得壮实。即使某些非核心生命周期发生问题，也不会造成系统性的问题。

再看看建筑，是一个空间的树状架构。看看人类社会，是一个树状架构。看看组织架构，也是一个树状架构。一个村庄、城市、国家，都是空间树状架构，树状架构无处不在，只要是有生命的地方就会有生命周期，就会有增长，就会有树。这是所有做架构的人都必须自觉认识到的。

谈到树，就必须要提分层，分层实际上就是架构树状拆分的结果。所以当我们采用分层的时候，心里先要有一个树的概念。大多时候，很多人直接采用分层的方式来设计，最后反而违背了树的原则，导致了很多复杂的问题，影响到系统的长大，得不偿失。比如跨层访问，就形成了图，违反了树的原则。

5.2 架构和树

当业务增长到一定的程度，新的瓶颈出现了。架构发生新的拆分，新的非核心生命周期从业务生命周期中拆分出来，但架构仍然是树状的，并没有发生变化。树根和树干仍然是核心业务来决定的，并没有变，只是架构树长高了，枝叶增多了。只要树根和树干没有变，树的品种就没有变。不同的业务，树干和树根就变得不一样了。虽然都是树，却是不同类型的树，比如松树和柳树是有区别的。

有人会问，不是树状的结构，算不算架构呢？严格来说，这不算架构，因为并不能够保证权责对等，业务会受其阻碍不能顺利长大，甚至导致业务的失败。

有时候不是树状的，业务也能够长大啊？从现象看确实是的，但实质是业务本身的增长掩盖了这一点。这种方式就像慢性毒药，往往很快就让业务到达瓶颈，进一步还会干扰业务拆分。

即使是上文说的树状架构，树到一定的高度是否也就到了极限？是的，但是这个高度是由业务本身决定的，由业务所生存的周边环境提供的能量多少决定的，而不是架构。架构的目的是要让树能够按照自己的规律长大，而不是在业务中加入架构自己的材料，干扰核心业务，这样反而会阻碍业务的长大。

架构是顺应业务生命周期规律的一种拆分，这个拆分始终是围绕着业务的核心生命周期的，结果也只有一种，就是一棵树。只有树的结构，才能够保证权责对等。只有权责对等才能够保证参与人能从其自身的工作中获利，才能够保障参与人能够持续不断地推进业务的生命周期，比如树叶自身也要长大，才能给树干供应更多的养分。没有这个激励机制的保障，任何拆分都是徒劳的，无法执行的。

大家可以从人类历史的规律中寻找线索，成功的比如商鞅变法、工业革命等，失败的比如王安石变法等等。这些都是人类有意识地对人类社会自身做架构改造的范例，由此也可看出架构落地执行的重要性。也只有树的结构，才可以保证权责对等，才能让不同人的努力汇集到树干，即业务本身的生命周期，让树长得更大。

第 6 章　概念

在前文中，我们讨论了什么是架构、生命周期等，这些基础概念对于做架构是非常重要的。大部分人对于每天习以为常的概念都自以为明白了，但实际上都是下意识的而不是主动的认识，所以当出现很多问题时也不清楚是怎么回事。

比如说“什么是桌子”，做培训的时候，我经常拿这个例子来问大家，大家的反应也很一致——这个问题还用问，不是很简单嘛，天天都在用。可是仔细一想，又都面带难色，认为自己知道，却找不到合适的语言来表达。继续深入思考后不同人得到的回答则又各式各样，各有各的侧重点。这实际上就导致了做架构的时候，不同角色之间的沟通会出很多问题，沟通的结果也就可想而知了。

人类架构实际上解决的是人的问题，而概念是人互相沟通并认识这个世界的基础，对概念的认识因而变得非常的重要。在这里尝试讨论一下如何去认识概念。当然这里不是讨论语言学，所讨论的和语言学可能大不一样。（如果大家对语言学感兴趣，也可以去了解一下。）

要讲概念，就要知道讨论都是以人的认识为主体进行的，属于人的认知。概念也属于人认识这个世界并用来沟通的手段，包括“概念”这个概念，也是一样的。在古代，不叫“概念”，称之为“名相”。

6.1　何为名相

一般我们认为：看到一个东西，比方说杯子，“杯子”就是一个名字，指代的看到的东西就是相，就是事物的相状。我们一听到“杯子”这个词，脑海里就会浮现出一个杯子的形象。而“杯子”这个词，是用来指代这个相状的，叫作名。合起来就叫作“名相”。

可是当我们把杯子打碎了的时候，我们还会称这个碎了的东西叫杯子吗？肯定不会，一般会叫“碎片”，如果我们把碎片磨碎了呢，名字又变了，叫作“沙子”。这就奇怪了，同样一个东西，怎么会变出这么多的名字？

6.2　究竟什么才是相

实际上“相”表达的不是一个具体的东西，如上面所提的一个瓷器杯子，并不是指这个瓷器，而是这个瓷器所起的一个作用：一手可握，敞口（一般不超过底的大小，太大口就叫碗了），并且内部有一个空间可盛东西。并不是指这个瓷器本身。这也是为什么人们从电视上看到一个人拿杯子的时候，就知道这个是杯子。但是实际上人们看到的都是光影而已，只是一个视觉作用。所以说相实际上代表的是这个影像，以及这个影像背后所产生的作用，是人对这个作用的认识，并不是具体的某个东西。而名是用来标识这个影像和作用，用来沟通交流的。

6.3　概念是沟通的基础

为什么需要给杯子的这个作用起一个名字呢？其实是为了解决“人需要一个可单手持握，而且希望避免直接接触所盛物体”这个问题。

所以说，每个概念实际上所解决的，还是人遇到的某个特定的问题，人们把解决问题的解决方案，给定了一个名字，这个名字就是对应的某个特定的概念。

对于概念这个词本身，为了统一指代这些名字，人们把起这类作用的名字称为“概念”。前文讨论的“架构”也是同样的一个特定概念，这里不再详述。

同样，什么是“建筑”？“建筑”实际上解决的就是“人需要独占的空间，并还能够比较方便地和外部世界沟通”的问题。这些概念反映的就是他们背后的独特作用。人们为什么一听就能够分辨出来不同概念的区别呢？这是因为人们的潜意识里对不同概念所代表的独特作用都有深深的体会。而当我们无法理解一个概念的时候，往往都是因为我们不知道，或者没有体验过这个概念背后所代表的独特作用。

再拿前面的“桌子”来举例，什么叫“桌子”？很多人回答，四条腿，或者说有腿，有一个平面，等等，但柜子不也是这样吗？为什么我们看到柜子，不会认为是桌子呢？即使有人把饭放在柜子上吃，人们看到仍然会问为什么在柜子上

吃饭？不会称其为桌子。如果明白了上面的道理，理解起来就很简单了，桌子实际上是为了解决人坐在椅子上，手还能够支撑在一个平面上继续开展活动的问题，一般会和椅子配对出现。坐在椅子上对着柜子工作有一个很严重的问题，不知道大家试过没有，就是腿无法展开。这么坐着超过半小时就知道是什么样的痛苦了。所以桌子的平面下方一定会有一个足够容纳膝部和小腿的空间，用来解决这个问题。解决了这些问题的装置，才能称之为桌子。

类似也可以定义出来椅子，由此可见，桌子和椅子的高度也是有限定的，都是用来解决人的问题，要符合人的身高：椅子的高度和深度，必须符合小腿和大腿的长度；椅背的高度要配合脊柱的高度；桌子的高度要配合小腿和脊柱的高度之和；成人和小孩的自然也就有区别了。这又变成人体工程学了。事实上要做好桌子和椅子，必须要理解人的生理结构，否则很难生产出来合格的桌子和椅子。正确的理解桌子和椅子的概念才能做好这一行。

现代人是坐在椅子上使用桌子的，而在古代，古人是席地而坐的，桌子的概念和现代不一样。可见正确地理解概念一定要了解概念所要解决的问题，这个问题往往也和当时的社会、文化、地域有关系。

同理，为何我们可以在不同的语言间进行翻译，是因为虽然语言不同，但是人类所面临的问题是一样的，只是所使用的名不同而已。

6.4 把握概念的力量

回过头来，根据架构的定义，要做好架构首先必须具备的能力就是要能够正确地认识概念，能够发现概念背后所代表的问题，找出核心生命周期。进而才能够认识目标领域所需要解决的问题，认识目标领域的核心生命周期，这样才能够为做好架构打好基础。事实上，这一能力要求在任何一个领域都是适用的。比如人们如果想要学习一项新的技术，如 Hibernate、Spring、Photoshop、建筑、音乐等，先去探索这些概念产生时所要解决的问题，学习这些新的技术或者概念就会事半功倍，快速上手；进入一个新的领域，也会非常地快速有效；使用这些概念来解释问题，甚至发明新的概念都是很容易的事。为什么强调这个呢？因为做架构的时候，很多时候都是在一个新的领域解决问题，必须要快速进入并掌握这个领域，才能够真正的解决问题。

通过以上几个例子，讨论了一下认识概念的误区，如何有效的去认识概念，明白概念背后的含义，以及如何利用对概念的理解，快速地进行学习。掌握了这些原则，会有利于我们在架构阶段快速地识别和定位问题，并能够高效率地进行沟通。

第7章　什么是抽象

在架构师的群体中，如果不谈抽象好像就不是一个合格的架构师。在讨论桌子这个概念的过程中，很多人会提出抽象这个概念，认为定义桌子实际上就是抽象的一个过程。因此有必要澄清一下抽象这个概念。

抽象这个词实际上是近代从日本舶来的，是对英文“abstract”的翻译，abstract是“抽取，摘要”的意思。中国古代没有使用过“抽象”这个词汇。

人们所见所闻，所思所想，只有人们自己才最清楚，体味最深。一旦人们想把这些个人的感觉传达给别人，就产生了沟通。语言是沟通中的一个方式。举个例子，一个人看到一朵红色的玫瑰花，非常漂亮，非常独特，非常感动。当他激动地尝试用语言“漂亮的红色玫瑰花”向朋友描述时，朋友脑海里浮现的仅仅是一朵普通的红色玫瑰花而已。他采用的正是这种抽取的方式，把他所见的认为重要的信息传达给朋友。可以看到，在这个沟通过程中产生了巨大的信息损失。为什么会是这样？

7.1　个性与共性

每个事物本身都有其独特的地方，我们称之为事物的个性。这朵玫瑰花的红是非常独特的，只有亲自看到的人才能够感受到它的魅力，这就是它的个性。一旦我们用“红色”来表述它，那么这个红色就变成了大家所共同认识的红色，这就是共性。而抽象只能够表达事物的共性，无法表达事物的个性。同样，在前面所说的“桌子”这个例子中，大部分人关注的也是共性，比如多少条腿，桌面形状等等。可是具备同样共性的事物有很多，怎么能从这些共性中推导出具备这些共性的一定是桌子呢？

所以要定义一个事物只能用这个事物独特的地方，也就是它的个性，而不是共性，而抽象恰恰主要关注在共性上，不具备这个能力。但要找出事物的个性，就必须要深入分析和理解这个事物的核心生命周期。核心生命周期明确了，事物的个性也就明确了。

比如还是这个“桌子”的例子，桌子的产生就是为了解决人类坐在椅子上可以继续开展活动的问题。一旦这个问题不存在，桌子的生命周期就结束了。比如古代的人们是席地而坐的，那时的桌子就很低。现代人改在椅子上坐，古代桌子的生命周期到现代就结束了。好像日本还保留有席地而坐的习惯，所以日本的桌子和中国的是不一样的，矮很多，这就是事物的特性。这个特性甚至会导致日本房间的高度都要比其他国家低，因为席地而坐的原因，导致头顶上方空间太大，居住的人会不舒服，必须降低房间的高度。可是日本的桌子和中国的有一点是一样的，桌子下面一定是空的，给腿留有伸展的空间，所以都叫桌子，这又成了不同桌子的共性，由人体的结构来决定：坐时一定会屈腿向前，因为人类的膝盖只能够向前弯。

很多人喜欢采用抽象来定义一个概念，特别是在软件行业，这个做法非常盛行。可抽象实际上是把不同概念的相似部分合并在一起，形成一个新的概念。这里面有很严重的问题：首先“相似的部分”在不同的人看来，并不一定那么相似，这会导致不同的人定义出来的概念不一样；其次，抽象之后形成的是一个新的概念，有着自己的个性，和原来那个概念所代表的个性并不一样，比如用容器来定义杯子、碗等。

因为所解决的问题不一样，两个概念的个性是完全不一样的，所以我们不能用抽象来定义一个事物。抽象实际上是一个分类的过程，抽取出事物中抽取人所关心的一些特征，并且不同的人抽取的侧重点是不一样的，甚至完全不同。

比如杯子和容器，很多人认为容器是杯子的抽象，但是实际上杯子是杯子，容器是容器，它们的个性，也就是所解决的问题是不一样的。当人们需要解决装东西的问题的时候，会说要一个容器；当人们需要解决单手持握且要装东西的问题的时候，会说要一个杯子。容器没办法代表杯子，只能够作为杯子的一个分类，杯子不仅仅只有容器这一个分类，还可以根据材质等其他性质进行分类。

7.2 个性是基础

所以大家想要做好架构，首先要具备识别个性的能力，也就是要有发现独特问题的能力。而个性是无法用语言表达的，必须亲眼来看看这朵独特的玫瑰，才能够体会其独特的红色，才能有共鸣。也就意味着，做架构的人必须亲自体验业务，感受业务，才可能真正认识业务的个性，真正认识业务所面临的问题。在理解业务个性的基础上，才能够谈共性。否则这个抽象就像是无根之木，局限于个人的主观认识，是很难长大的。

第 8 章　识别问题

根据前面介绍的架构的定义，做好架构首先需要做的就是识别出需要解决的问题，也就是核心生命周期。一般来说，如果把真正的问题找到，把核心生命周期找到，问题就已经解决八成了。因此可以说，识别问题这个能力基本上就决定了架构师的水平。

8.1　面对问题有哪些困难

我们先看一则笑话。一位女士对老公说：把袋子里的土豆削一半下锅。结果所有土豆都下锅了，并且锅里煮的每一个土豆都被削了一半皮。

可能很多人会认为这只是沟通的问题，然后一笑了之。其实，之所以出现类似的问题，是由于人们大多数时候都在急于解决问题，担心自己的时间不够而焦虑，恨不得马上完成别人交给自己的工作，尽可能地减少问题对自己时间的占用，而不关心“真正的问题是什么”。还有一个有意思的现象，看到这个笑话后大家都会会心一笑说，这个老公十有八九是程序员吧。

当我们去解决一个问题的时候，一定要克服对时间的焦虑，耐心地先把问题搞清楚。去看看软件开发工作者的时间分配也可以看出，大家把大部分时间都花在了讨论解决方案和实现的细节上，基本都不会花太多时间去思考“问题是什么”，“我们在给谁干活”。或者即便想了，也是一闪而过，快速地凭自己的直觉做判断。就如前面章节讨论“桌子”的情况一样，美其名曰“抽象”，其实只不过是想省力省时间罢了。只有真正敢投入思考“是谁的问题”，关心“真正问题是什么”的工程师，才有可能成长为真正的架构师 。

在人们处理问题的时候，一般会面临以下几种情况。

A. 被告知要解决一个问题，但是交过来的实际上是一个解决方案，而不是问题本身。此时往往容易不经过思考就直接去执行了，那么最后肯定是对方并不满

意，导致反复修改。

B. 被告知要解决一个问题，此时往往急于解决问题，未经过仔细分析，通过直觉或网上搜索就找到了一个解决方案，然后马上考虑该解决方案如何落地。或者有几种解决方案，自行挑选一个自认为比较合适的，还为自己的反应快而沾沾自喜。然后做出来对方不满意，仍然导致反复修改。

8.2 如何识别问题

对于上述情况，怎样才能尽可能地一次性发现问题并解决问题呢？我们来尝试分析一下。

沟通中大家基本都会面临一个很大的问题：缺乏主语。而大家都心照不宣地忽略这个主语，沟通的时候互相之间也都以为能懂得对方所说的主语是谁，结果大家都一起犯错误，出现问题后都认为自己有道理，互相指责。比如前面笑话的例子："把袋子里的土豆削一半下锅"，主语是谁？

识别问题的一个最重要的前提就是要搞清楚：是谁的问题，也就是要先明确问题的主体是谁。主体搞清楚了，问题的边界也就跟着确定了，再去讨论问题才有意义。

以切土豆的例子来分析。

（1）女主人提出一个问题：要削土豆下锅煮。

（2）男主人面临一个问题：女主人下达了一个自己必须要完成的任务。

每个人都是优先处理自己的问题，男主人自然选择了2，赶紧完成这个任务，这也是大部分软件工程师处理问题的方式。这时所面对的已经是一个解决方案了，削一半吗？男主人马上以自己认为正确的方式解决了自己的问题。看上去没什么不对啊，也难怪能得到大家的共鸣。那么这里面犯的错误是什么呢？

（1）女主人给出的实际上是一个解决方案，而不是煮土豆这个问题本身。女主人当时亲自执行这个解决方案可能有时间上的困难，就把执行解决方案作为一个任务，委托给了男主人，形成了分工。

（2）男主人得到了一个任务，尽心尽职地把这个任务完成了。

最后的结果是什么呢？女主人会认为男主人理解能力差，抱怨男主人思路有问题，没有生活常识。男主人则认为自己按照女主人的要求执行了，并且觉得自己做的是对的，因此会抱怨女主人没把问题讲清楚，自己只是奉命行事。真正的问题被忽略了，自然也没有被解决，最后大家还得一起收拾残局。不但要继续解决原来的问题，还要解决新创造出来的问题，结果反而事情越做越多了。归根结底，其原因在于男主人太专注于解决自己的问题，想节省时间，反而导致工作没有完成，还浪费了更多的时间。类似的事情每天都在我们周围发生，如果没有发现问题的根本原因，那么下次还会继续发生。

把原因归结为沟通问题也是可以的，但这对于解决问题似乎并没有太多的帮助。搞明白目标问题“是谁的问题，是什么问题”，当然也是需要沟通的。为了帮助自己更快地搞明白问题，大家首先要做的事情是问正确的问题。接到任务马上就去开工实现，并不能加速问题的解决。架构师应该问的第一个正确的问题就是：目标问题是谁的问题。在这个故事中，男主人要解决的，实际上是这个家庭晚餐需要吃土豆的问题，目标问题的主体实际上是这个家庭的成员。

明白了问题的主体，这个主体就自然会带来很多的边界约束。比如土豆是要吃的，是要给人吃的，而且还是要给自己的家人吃的。“削土豆下锅”这个问题，因为识别了问题的主体，自然而然地就附带了这么多的隐含信息。挖掘出隐含信息，就自然而然能够问出来其他问题了，比如后续如何煮，是否放高压锅煮，放多少水，煮多长时间等，说不定还能够识别出来女主人给的这个解决方案可能是有问题的。这个时候才算是真正地明白了问题。可以想象，这样下去最后的结果一定是大家都满意的，因为真正的问题被发现并解决了。人们只有真正明白了别人给过来的问题是谁的问题，才能够完成自己的任务，把自己的问题解决掉，才能真正地节省自己的时间，而不是反过来。

当我们处理问题的时候，如果发现自己正在致力于把自己的工作完成，就要马上警惕起来，因为这样下去会演变成没有主人翁精神（Ownership）的工作态度。在面对概念的时候，也会不求甚解，最终会导致无法真正的理解概念。作为软件工程师或者架构师，我们大部分时候是要去解决别人的问题，“别人”是谁，是值得好好思考的。

从上面的分析可以看出，找出问题的主体，是做架构的首要问题。一定要明

白，架构师要解决的问题都是人的问题。更进一步，架构师要解决的基本都是别人的问题。别人的问题解决了，架构师自己的问题才能解决掉。再进一步：

任何找上架构师的问题，绝对都不是真正的问题。

为什么呢？因为如果是真正的问题的话，提问题过来的人肯定能够自己解决了，不需要找架构师。

架构师都要有这个自觉：发现问题永远比解决问题更加重要。

8.3 寻找问题主体

当问题的主体离架构师越远，就会让找出问题主体的过程越加困难。再举一个软件行业都比较熟悉的例子：用户向产品经理提出要求，想要一把锤子。这就是典型的把解决方案当作问题的。真正有问题的主体是谁，是用户、设计师还是施工队？如果产品经理当成是自己的问题，那么毫无疑问就会给用户一把锤子了。产品经理需要识别：用户究竟是二传手，还是问题的真正主体。如果主体是设计师，那么问题的边界就变成了设计师的问题，如果主体是施工队，那么问题就变成了施工队的问题，如果主体是用户，那么就要看看用户到底有什么困难，绝对不是要一个锤子这么简单。这也说明了，确定问题的主体对于确定问题的边界有多么的重要。

明白了问题的主体，人们才可能真正地认识问题是什么。因为问题的主体是问题的隐含边界，边界不确定下来，问题就是不确定的。一旦确定了问题的主体，剩下的就是去搞明白主体有哪些问题。这就比较直接明了，常用的方式就是直接面对主体进行访谈，深入到主体的工作生活当中，体验并感受这些问题，识别出关键生命周期，进而识别出非关键生命周期，进而在这个基础上做架构就非常容易了。

一般来说，从问题暴露的点，一点点地去溯源查找，一定会找出来究竟是谁的问题，以及是什么问题。最坏的情况就是我们时间或者能力有限，实在是无法定位出是谁的问题，比如系统出故障，也就意味着我们无法根本解决问题时，最好的办法就是去降低问题发生所带来的成本，尽量去隔离问题所影响的范围，给自己争取时间和空间去识别真正的问题。

总结一下，要想正确的认识问题，需要问两个问题：

（1）这是谁的问题？

（2）有什么问题？

当得到的回答是支支吾吾的时候，大家就要知道需要在这里花上足够的时间来明确问题，此时要避免把时间浪费在其他的地方。一般来说，寻找“是谁的问题”会花比较多的时间，也是支支吾吾最多的地方，因为架构要解决的问题都是人的问题，而大部分人总是忽略了主语。一旦确定了答案，“是什么问题”就会变得相对容易，因此架构师的能力大部分会体现在对“是谁的问题”的识别上。

第 9 章　切分的原则

前一章讲了如何识别问题，在识别出是谁的问题之后，大家会发现，在大部分情况下，问题都迎刃而解，不需要做额外的动作。很多时候问题的产生都是因为沟通的误解，或者主观上有过多不必要的利益诉求导致的。但总还是有一部分确实是有问题的，此时需要做调整，那么就必须要有所动作，以进行相应的解决，而解决的方式就是架构切分。

9.1　切分就是利益的调整

架构师要非常清楚，所有的切分调整，都是对相关人的利益进行调整。为什么这么说呢？因为维护自己的利益，是每个人的本性，这是在骨子里面的，每个人都不能逃避这一点。

随着社会的发展，产生了人类的分工，分工背后的动力来自于每个人寻求自己的利益最大化的冲动。人们都希望能够在最短的时间内，得到更大的产出，把自己的利益最大化。所谓的利益，其实就是保障自身的生命周期活动推进的质量。比如，在同样的时间内，生活得更舒适，更轻松，更安全，占有并享用更多的东西。但是每个人的时间和能力都非常的有限，不可能什么都懂、什么都会。因此人们需要放弃一些自己不擅长的事情，用自己擅长生产的东西去换取别人擅长生产的东西。对比一个人干所有的事情，分工的结果让大家都能够得到更多。也因此产生了一个互相依赖的社会，谁都离不开谁。这就是人类社会自然而然产生的架构切分，背后的原动力就是人们对自己利益的渴望，对提高生命周期推进质量的追求。人们对自己利益的渴望也是推动社会物质发展的原动力。

比较有意思的是，在这个模式下每个人都必须要舍掉自己所拥有的东西，才能够从别人那里得到更多的东西。有些人不愿意和别人进行交换，不想去依赖于别人，这些人的生活就很明显会差很多，也辛苦很多，很自然地就被社会淘汰了。

如何在这个社会上立足，判断的标准就变成：如何提供更好更有质量的服务给这个社会。提供的服务更好更多，就能够换取更多更好的生活必需品。

这也是人们在人类社会生存、立足、做人的基本道理和原则：先要付出才能够有收获。

9.2　为什么需要切分

当人们认识到要主动地去切分一个系统的时候，就不能忘掉利益这个原动力。所有的切分决策都不能够违背这一点，这是大方向。结合前一章“识别问题”，一旦确定了问题的主体，那么系统的利益相关人（现代管理学语言中叫 Stakeholder）就确定了下来。所发现的问题，基本会有两种：

a. 某个或者某些利益相关人时间或空间上的负载太重。

b. 某个或者某些利益相关人的权力和义务不对等。

9.3　切分的原则

利益相关人负载太重是导致切分的原因，权责不对等是切分不合理而导致的新问题。权责不对等而导致的新问题，最终还是要回到利益相关人的负载太重来解决。对于负载太重的问题，本质上都是时间上的负载过重，有限的时间内无法得到期望的产出。每个人的时间是有限的，怎么在有限的时间内做出更多的事情？只有把该利益相关人在时间上连续的动作，切分成时间或空间上可以并行的动作，在时间或空间上横向扩展，让更多的人并行工作来提升产出。

要进行切分的话，自然就要遵循架构切分的原则：

（1）被切分的生命周期，如果必须要生命周期的主体在连续时间内持续执行，而且不能够被打断并更换生命周期主体的话，就不能切分出去。这类生命周期是切分的最小粒度，受限于当前的技术水平无法进一步切分。比如孕妇怀孕，必须要十月怀胎，不能够切分成十个人一个月完成。某个生命周期能否被切分出来，完全取决于当前技术发展的水平，比如试管婴儿技术迎来更大的突破，可能就不再需要妈妈十月怀胎了，那时切分说不定就成为可能。

（2）每个生命周期的负责人，对所负责生命周期的权力和义务必须是对等的。

比方说妈妈十月怀胎生小孩，妈妈天然有权力处置小孩的出生和抚养，同样也对小孩的出生和抚养负有义务。为什么必须是这样呢？因为权力和义务要是不对等的话，就无法形成树状架构，此时切分出来的生命周期就很难组合回去，整体花费的时间更多。另外该生命周期的负责人如果无法从生命周期的活动推进中获利，就违反了分工交换的原则，这导致每个个体的利益受到了伤害，最终切分出来的生命周期无法顺利地执行。切分出来执行的效率如果比没有切分出来还要低，最终会导致整体的利益也受到了损害，这违背了分工提升整体利益的初衷。权责不对等最终会导致架构无法落地，甚至导致业务失败，这一点从人类社会的发展变革中也可以看出来，典型的如商鞅变法、王安石变法、工业革命等。

（3）切分出来的生命周期，不应该超出一个自然人的负载。当然每个人的能力不同，负载能力也不一样，社会技术水平不同，负载能力也会有区别，需要不断根据当时的实际情况进行调整。这也就是生命周期的运营。切分出来的只能是非核心生命周期，其主体和核心生命周期可以不一致。并且这些非核心生命周期必须是完整的，也就是内聚的。

（4）切分是内部活动，内部无论怎么切，对整个系统的外部都应该是透明的。如果因为切分导致整个系统解决的问题发生了变化，那么这个变化就不属于架构的活动，或者说这个切分是不符合业务的。比如一棵松树不能突然长出一朵玫瑰花。当然当我们把问题分析得比较清楚时，整个系统的边界会进一步地完善，这就会形成业务的螺旋式进化，也就是业务生命周期的调整。但这不属于架构应该解决的问题。业务进化的发生，也会导致新的架构切分，新架构和原来的架构并没有关系。抛不开原来的架构的影响，势必会导致新架构陷入困境。

9.4 树和分层

权责对等的原则（切分原则 2）确保我们不能违反人性。因为维护自己的利益是每个人的本性，只有权力和义务对等才能保证个体的利益。从权责对等的原则也可以推理出，所有的架构拆分都应该形成树状的结果。不应该变成有向图，更不应该是无向图。很多人一谈架构就必谈分层，但基本上都没有意识到，分层的前提是把一个生命周期的连续过程拆分成了一棵树，因为有了树才有层的出现。从根节点下来，深度相同的是同一层。树和层是数学概念，这里就不展开了，感

兴趣的可以去复习一下数学或算法。对很多人来说谈架构就是谈分层，这似乎也没有错，但如果不知道层的背后是树，而只用层的方式处理，最终会破坏树的结构，得不偿失。

同样，对于一个组织架构，也可以做一个粗略的判断：如果一个企业的组织架构出现了“图”，比方说多线汇报，一定是对利益相关人（Stakeholder）的利益分析出现了问题，必定会导致权责不对等的情况发生。权责不对等的情况一旦出现，架构师必须要马上意识到，这个问题持续时间越长，企业的运作效率就会受到越大的影响，对利益相关人的利益是非常不利的，同样对于企业的利益也是不利的。必须快速调整利益相关人的职责，消除权责不对等的问题，让企业的组织架构成为一个完美的树状，并且使树的层数达到尽可能的低。树层次越深，沟通损失就越大，效率也就越低。平衡树的深度可以做到比较均衡。

如果某个节点的能力很强，可以帮助减小树的高度；技术的提升也可以提升每个节点的能力，降低树的层数。很多管理学都在讨论如何降低组织架构的层数，使得管理能够扁平化，原因就在于此，我们在这里也不展开讨论。

从这里可以得出一个结论：一个好的组织领导也一定是个很好的架构师。

9.5　切分与建模

架构切分的过程就是建模的过程。在早期流量不大的时候，往往一个人同时会兼很多工作，一个系统往往也会做很多的事情。在流量增大后系统很快就达到瓶颈，这个时候就不得不进行拆分了。要做拆分就必须要去识别核心生命周期和非核心生命周期，把非核心生命周期切分出来。切分出来的不同子生命周期就形成了不同的概念。所以每个概念背后就是一个生命周期，每个生命周期都是一个模型。核心生命周期模型把所有的模型通过树组织起来，形成一个新的模型。

这些不同的概念，大部分时候人们已经自发地建好了，架构师更多地是要去理解这些概念，识别概念背后所代表的问题，以及该概念对应的人的利益。

比如人类社会按照家庭进行延续，形成了家族；不同家族由于共享一片土地资源，慢慢形成了村庄和村庄联合体；不同地域结合，形成了国家，等等。由于利益分配的原因，形成了政权。每次政权的更迭，都是由不同群体的利益重新分配的动力所推动并决定的。

对于一个企业也是一样的，一开始一个人干所有的事情。当业务量逐渐变大，很快就超过了一个人能够处理的容量。企业的创始人就要开始思考企业的核心生命周期，尽可能地让企业生存更久。其他的非核心生命周期就会被分解出来——开始招聘人进来，让他们围绕着核心生命周期，通过树的方式组合在一起，完成企业的日常事务。整个企业的事务，就形成了很多的子生命周期，切分出来了很多新的概念，如营销、售前、售中、售后、财务、HR 等。

此后，企业创始人的工作就变成了如何保证核心生命周期的运转，以及组合这些不同的概念完成企业的工作。如果业务再继续增大，这些被切分出来的部分还要继续发生拆分，拆分时仍然要识别该部分自身的核心生命周期，并按照前述的原则切分出非核心生命周期，树的层数逐渐增加。如果某个技术的提升，提高了某个角色的生产力，使得某个角色可以同时承担更多的工作，就会导致多个职责和权力合并为一个，降低树的层数。如果业务发生了收缩，也会发生合并的情况，降低数的层数。

9.6 切分的输出和组织架构

架构切分的输出实际上就是一个系统的模型，一棵以业务生命周期为树干的树。对于这棵树，要识别出有多少个相关方，每个相关方需要承担哪些权力和义务，不同的相关方是以什么方式组合起来完成整个系统。有的时候是从上往下切，有的时候是从下往上合并，有的时候两个方向皆有之。

架构切分的结果最终都会体现在组织架构上，因为架构的切分是对人利益的重新分配。另一方面，架构切分需要组织架构来保障实施。负担重的相关利益人要减轻职责和权力；负担轻的相关利益人要增加职责和权力；所有人负担都很重，就要增加人，形成新的架构切分，或者引进新的技术，提升大家的生产力，以形成新的架构切分。所以进行架构切分的时候，往往也就是组织在长大的时候。

从这方面也可以看出，任何架构调整都会涉及组织架构。同样，如果对于利益相关人的利益分析不够透彻，也会导致架构无法落地。因为没有人愿意去损坏自己的利益，一旦去强制执行，人心就容易涣散。当然，这也不一定是坏事，只要满足权责对等的原则，就能够建立一个很好的新次序和新利益关系，保持组织的良性发展。长久来看这是对所有人都有益的，虽然短期内对某些既得利益者会

有利益损害。

总结一下：

1. 架构的切分的导火索是人的负载太重，也就是时间不够。

2. 架构的切分实际就是对利益相关人的利益进行切分或合并，使得每个利益相关人的权责是对等的，每个利益相关人可以为自己的利益负责。

3. 架构切分的最终结果都会体现在组织架构上，只有这样才能够让架构落地并推进。

4. 架构切分的结果一定是一个树状，这也是为什么会产生分层。层数越多沟通越多，效率越低，分层要越少越好。尽可能变成一棵平衡树，才能让整个系统的效率最大化。

一个好的架构拆分，会形成一棵树，慢慢会长成一片森林。每棵树在这个森林里都能够获得所需要的养分，有自己的空间。每棵树的内在是平和的，舒展的，遵循自己的生命周期规律，顺其自然地长大，一派欣欣向荣的景象。这就是架构的魅力所在！

第 10 章　架构与流程

看到这里大家会发现，生命周期分析贯穿了整本书。是的，生命周期像是滚滚东流的水一样，不断地流去、积聚，一去而不复返。而架构要做的就是让不断增长的水流能够顺利地长大，而不要变成洪水，冲毁一切。

10.1　什么是流程

在做生命周期分析的时候，避不开的一个概念就是流程。流程是什么呢？流程就是多个角色为了把一件事情做好，按时间顺序协作并完成的整个过程。这其中包含了很多的活动，很多的判断，很多的分支。这是不是和生命周期很像呢？确实是的，一个流程的执行过程，本质上就是一个事物的生灭过程，是一个生命周期。

但是流程关注的并非是生灭。流程关注的是多个人如何把同一个目标完成。因为涉及到多个角色，那么在一个事物发展的过程中，就会有很多种可能性，每个可能性下还有分支。因此，流程是描述整个事物发展各种可能性的树。

在对业务架构分析的过程中，对业务的流程梳理是非常重要的事，往往就代表了业务的核心。但是流程梳理和分析的误区很多，往往把人带入了细节之中，忘记了流程本身的目的。最后导致的结果就是，流程淹没在用户的访问代码中，每次的修改都是伤筋动骨，成本非常的高。

就好比人们看一棵树一样，大家总是注意到丰富多彩的树枝、树叶、花朵，而往往看不到更重要的树干、树根。人们总是容易把最重要的事情忽略，关注那些无关紧要但是很吸引人的东西。要知道，树干树根结实了，自然会长出好看的花朵。同样，在流程分析中，人们往往会被这些流程分支迷惑了，忘掉了流程背后的实质，还是业务的核心生命周期。

10.2　流程和架构拆分的关系

流程的核心是业务的核心生命周期。如果没有分工，是不会产生流程的，一个人按时间顺序从前做到后，没有人与人的协作。当业务的生命流程发生架构拆分之后，由一个人完成业务生命周期所有的活动，变成由多个人合作来完成。可是业务生命周期按时间顺序推进的特性并没有变，整个事情也并没有变，所以流程就把不同职责的角色串联起来，完成业务生命周期的所有活动。也就是说，流程的推动都是和人相关的，流程实际上串联出来的是不同角色协作的过程。

那么流程的制定是掌握在谁手上的呢？掌握在流程的创立者手上，他也往往就是发生架构拆分的那个角色。比如，对某个角色所负责的某个生命周期进行拆分，该角色就把自身的很多非核心生命周期活动，下放到了自己所负责组织架构的其他不同角色身上。拆分后该角色还是负责该生命周期，并没有改变，但是执行生命周期活动的人变多了。而为了把拆分后的其他角色的工作串起来，该角色就必须按照原来的生命周期建立流程，组合其他角色的工作，完成原来的生命周期活动。

生命周期拆分之后，就意味着权力的下放，因此往往会产生审批的流程。审批流程则是确保权力执行的工具，因为审批代表了架构的拆分和权力的下放。比如审批流程链条的最后一位角色，往往就是创立审批流程的人。再比如文件的审批流程，实际上就是组织架构树上，从底层节点，一层一层往根节点审批的过程。由于组织架构是一棵树，所以流程实际上就是对流程负责人所负责的组织架构树某种方式的遍历。

也就是说，核心是业务生命周期。业务为了长大，发生了架构拆分，形成了一棵树。而流程则把拆分之后的业务进行组合，通过遍历树，重新形成了业务生命周期整体。

第 11 章　什么是架构师

架构师始终是一个比较神秘的角色，就像架构一样，好像也没有一个定论。每个人心中的架构师都是不一样的，并且有一个规律，都把自己搞不定的事情交给架构师，以为架构师就是能搞定自己搞不定的事情的人。那什么是架构师呢？

11.1　架构师做什么

做什么事决定了一个人是什么人。为什么我们称某个人为架构师，肯定是因为他在做可以被认为是架构师的事。那么哪些事是架构师应该做的呢？从前面我们所探讨的什么是架构可以看出：

架构的目的就是为了增长。

而要达到增长，就必须要把很多人合并起来做同一件事情，并且使他们做的事情合并起来达到 1+1>2 的效果，最少也要达到 1+1=2 的效果。而在现实生活中，人数增长到了一定的程度，沟通效率就会下降。到了一定程度，人越多产出反而越少。这个时候就需要架构师。

架构师会把需要增长的业务了解清楚，挖掘出核心生命周期，并确定核心生命周期的主体。换句话说架构师要发现问题的主体，并确定核心问题。在确定业务核心生命周期以及核心生命周期的主体之后，架构师还需要对业务核心生命周期进行分析，剥离出非核心生命周期，并根据当前人员的状况，合理地分配非核心生命周期的权责。这样不同的人就可以并行地互不影响地做不同的事情，最后根据核心生命周期，把他们的工作成果组合起来，达到 1+1>2 的效果。

以上仅仅是把现在的问题解决好，还需要更进一步，那就是根据对不同生命周期的运营情况，对未来的增长做一定的预判，提前做好规划，做相应的人员、技术的储备——这就是战略架构。

11.2　架构师也是人

通过前面描述的所做的事情来看，架构师并不神奇。架构师也是人，也和大家有一样的核心生命周期：生、老、病、死，也同样需要吃饭喝水，背后也有一个家庭。为了支撑自身的核心生命周期，也需要参与到社会分工当中去，通过付出劳动来获取所需的生活必需品。

架构师独特的地方在于，他自己的社会分工职责在于给别人分工。有一个“分粥效应”的故事：一群人分粥，如何才能够分得公平？如果架构师也是从同一锅中分得粥的一个，架构师怎么能够一直确保大家都能够分得公平呢？很多公司在使用架构师的时候，都产生了这个问题。是不是轮流做架构师才好？

问题的关键在于，架构师作为分粥者所得到的粥不能来自于和大家相同的一个锅，而应该是在另一个锅里，由解决问题的结果来决定得到粥的多少。也就是说，架构师的权责也是要对等的。因为工作的时候并不是说每个人的粥都一样多才好，有些人能力强，有些人能力弱。架构师工作的反馈，应该由问题的解决效果，也就是增长的效果来决定。如果以很少的资源增长达到了很大的业务增长，肯定是一个非常好的架构师，所节省下来的资源应该回馈给架构师，形成正向反馈。如果能够长期持续地做到这一点，那么一定是一个非常顶级的架构师，成为顶级架构师的过程中，少不了持续的正向反馈。

从这一点来看，架构师要对总体增长的效果负责，那么自然意味着要能调动资源，也要有权力分配。所以架构师往往都是某个业务或者领域的负责人。如果不是负责人，那么他也只是从同一锅粥中分得粥的一位，此时我想他是做不好架构师工作的，因为他也只是一个叶子节点，他的分工方案很难得到落地。

11.3　人人都是架构师

架构师是一个职业吗？很多人心里都有这个疑问。

架构实际上是一种通用的行为，只是大部分时候人们自己没有意识到而已。比方说，很多人的办公桌看起来乱七八糟，但是他的工作非常有效率。因为他在长期的实践之后，根据自己每天的工作，整理出了自己工作的核心生命周期，与其相关的每样东西都摆放在合理的位置，其他的东西都放在非关键路径上。这个时候他已经在做架构思考了，因为他已经在专注于自己工作的核心生命周期，把

非核心的都尽量剔除掉，以提高自己每天的产出。他已经是自己每天工作的架构师，只是他自己没有意识到而已。任何一个人，只要他在思考如何做得更好，如何做得更多，必然会导致他去思考核心生命周期，那么他已经在做架构的思考了。所以人人都是架构师，人人都具备做架构师的能力。

相反，如果人们不去切割自己的核心生命周期和非核心生命周期，追逐不恰当的方向，往往会陷入时间的焦虑，甚至会导致生活质量的下降。

11.4 架构师和权力

做好架构师，不是每一个人都能够办得到的。因为做架构师，必须要具备调动资源的能力。比如我们总说“改革开放的总设计师”，这个架构设计谁都能做，可是没能力调动资源，就无法落地，这个架构设计是无效的。只有具备权力落地的人才能叫架构师。也就是说，架构不仅仅是设计的层面，还有执行的层面，并且执行的层面是决定性的。所以往往架构师都是以领导的姿态存在。说到这里，大家往往会拿建筑来说事，建筑哪有专门的“架构师”头衔（Title）啊？大家可能都没有注意到，建筑的架构师实际上就是一个建筑项目的领导，他能掌控生杀大权，正是因为他的头衔的本质就是“架构师”。

所以架构师实际上就是权力的代名词，在人类社会，具备权力往往就是一个群体的领导者，这些领导者都是架构师，只是没有架构师的头衔罢了。当然很多行业有“架构师”的头衔，但是并不具备权力，所以实际上不是架构师，只是一个建议者的角色，相当于“军师”。也就是说并不是叫“架构师”的都是架构师，也不是能做设计的都叫架构师，也并不是所有的架构师都有架构师的头衔。

架构师有权力的同时也有其义务，也是权责对等的。架构师要时时把增长放在自己的第一考虑要素，把识别核心生命周期及其主体作为第一思考，这样才能确保权力的合理分配，保证增长的效率。这是真正的架构师和普通人思考的区别。要做到这一点，架构师必须深入到业务的核心领域，亲身体会业务的痛点，业务的个性，这样他才能够挖掘出业务的核心生命周期，才能够做好拆分，才能够做好架构设计，做出的设计才是具备落地可能性的。

我们总说大自然是一个很好的架构师。比如，太阳时刻在燃烧自己，为整个太阳系带来光和热，成为地球生命的发动机，地球因此而万物峥嵘。地球上的每

个生命都因此而能遵从自己的生物本能，从生走到灭，为这个世界带来一系列改变，然后消失得无影无终。架构师也要向大自然学习，向人类社会学习，从中汲取营养，进而通过合理的架构释放所服务领域的增长动力，让业务能够顺利长大，使得人们的生命周期有更多的意义，就好像生命被延长了一样。

第二部分

软件架构

探讨软件的本质，进而讨论架构在软件中的作用，
并与建筑架构做一定的对比

第二部分
软件架构

探讨软件的特质，进而讨论架构在软件中的作用，
并与建筑架构做一定的对比

第 12 章　什么是软件

前面整体探讨了一下什么是架构，以及如何做好架构等必要的概念。这些概念对于各种不同的领域都是有用的，希望它们能够给大家带来一定的启发，也希望大家能将其应用到自己所处的领域中。接下来我们把这些思考应用在软件中，探讨软件以及软件架构，以及如何更好地设计和实现软件。

12.1　以模拟人为目标的冯·诺依曼结构和图灵机

软件的历史，可以说是用机器模拟人的历史的进一步发展。在计算机出现前，人们用机器来代替人进行生产，也就是所谓的机械化生产。软件的出现，也许很多人没有意识到，包括这个历史过程中的参与者，实际上是人类有意无意地在计算机上模仿自己这种原始动机的体现。某种程度上和前文分析的一样，人们对时间的恐惧，导致了人们延长自身生命的努力，提升自己的生产力是其中的一种途径。而软件则让人们能够节省大量的工作时间，有更充足的时间去关注并推进自身的核心生命周期。

从冯·诺依曼结构开始，程序逻辑开始脱离硬件，采用二进制编码。加上存储，配合输入输出等硬件，一个简化的大脑就出现了。图灵机则是模拟大脑的计算，用数学的方式把计算的过程定义了出来，著名的邱奇-图灵论题：一切直觉上能行可计算的函数都可用图灵机计算，反之亦然。软硬件两者一结合，一个可编程的大脑出现了，这也是现在为什么我们把计算机叫作电脑的原因。在硬件上运行的程序就是软件，用来控制硬件的行为。通过软件的编写，可以制造出各种各样具备不同能力的“人”，而所花的时间比培训一个真正的人少，因为软件不用吃饭，只需用电即可。

12.2　成本为王

在软件发展的初期，软件直接采用二进制编写，从硬件到软件成本都非常的高。随着半导体技术的进步，硬件的成本越来越低，性能越来越高，甚至出现了摩尔定律：当价格不变时，集成电路上可容纳的元器件数目，约每隔 18～24 个月增加一倍，性能提升一倍。软件方面，为了简化难度，开始采用汇编语言，进一步出现了类似于人类语言的高级语言，比如 C/C++/Java 等，这使得人类可以用类似于人的语言，把人类的知识传递给计算机，训练计算机掌握某种技能。

在软件行业，训练计算机的人，也就是编写软件的角色，叫作软件工程师。随着计算机的运用越来越广泛，软件工程师越来越多，开发软件的成本也越来越低。计算机就好像是一个只需要电而不需要休息的人，可以无休无止地工作。人们越来越愿意把原来只有人才能做的事情，交给计算机来做。结果就导致软件越来越丰富，能够做的事情也越来越多，成本也越来越低。成本是我们为什么采用计算机和软件的主要动力，它可以帮助人们节省大量的人员培训，减少雇员的数目，使得人们脱离重复的劳动。

限于技术的发展，软件可以模拟的业务还非常有限，软件行业的门槛也很高。企业在采用软件之前，需要先搞清楚软件可以帮助企业做什么。软件的出现，确实可以降低业务的成本。但是在没有软件的情况下，业务也是一样能跑的。如果只是为了跟风要采用软件，说不定反而提高了成本。我们经常说软件或技术（软件当然也是一种技术）是业务的使能者（Enabler），实际就是把业务的成本从原来很高降到了很低的程度而已，并不是有了什么新的业务。另外，软件也不是降低业务成本的唯一方式。

12.3　天空才是极限

再看看人类，受限于所能够获取的能量和地球的引力，大部分人的身高都无法超过 2 米，寿命只有 100 年左右，可工作的时间只有 40 年左右。即使这 40 年都在工作，按每周工作 40 小时算，这 40 年只有大约 1/5 的时间在工作，每天还得吃饭睡觉。计算机则不一样，计算机的能量来源是电力，只受限于电能。只要资源允许，计算机可以长大到“天空才是极限”，从互联网的发展就可以体会到这一点。

在机器化生产的时代，机器也是靠电能的，可是机器的编程是在工厂生产的时候就已经确定了。机器的生产成本很高，而且生产出来之后功能基本就无法改变了。计算机其实也是一种机器，但计算机和以往的机器不同，计算机的功能反而是在出厂之后，人们通过编写软件再确定的。也就是说，软件的出现使得机器的生产生命周期产生了分工，形成了硬件生产生命周期和软件生产生命周期两种形式。

软件的出现，让只有“身体”的机器具备了“大脑”。机器通过更新“大脑”中软件的方式不断地学习，变成了一个“活着”的虚拟“人”。虚拟人的出现，导致人类社会也开始软件化、互联网化。

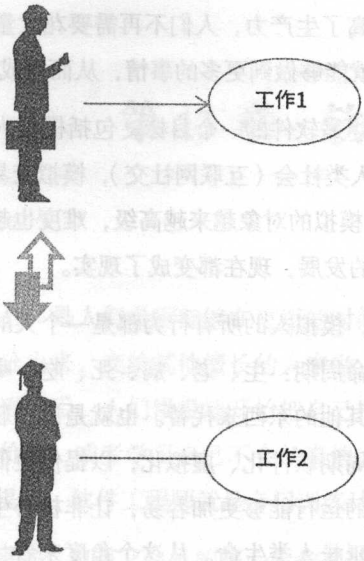
原来必须由实体店来进行售卖的东西搬到了互联网上售卖，开店成本更低，并且能够让更多的人访问到所售卖的商品。想象一下，物理上一个门店受限于道路、建筑等空间的大小，容纳的人流是有限的。在现实生活中，每天的人流量如果达到百万级别，那么人工创造的场地是很难承受的。但是在互联网上，访问量千万级别都不算什么，因为电量才是它的极限。

最终的结果就变成，每个人能够负担的工作越来越多，成本越来越低。人类的生命也相对地增长了，因为可以花更少的时间做更多的工作。这也是为什么软件行业这么热的原因。

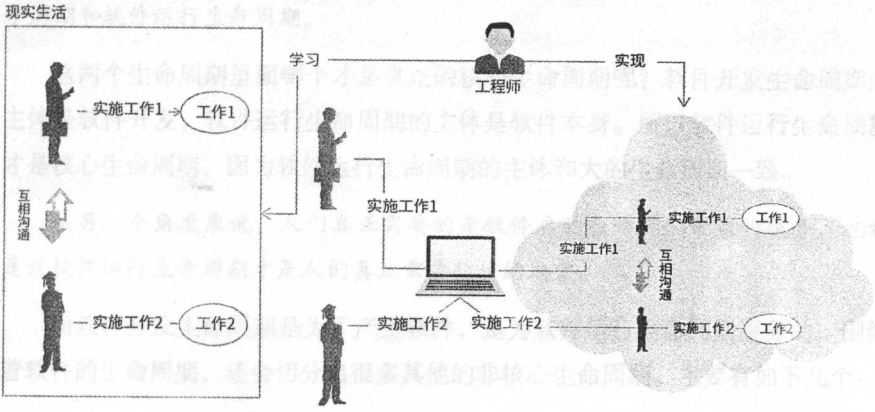
随着软件规模的变大，做好一个软件也变得越来越难。早期的程序员写程序，主要是为了帮助自己研究课题。这些程序员熟练了之后，在提高自己生产力的同时，发现还可以帮助别人写程序。慢慢软件就变成了一个独立的行业，服务于各行各业。程序从早期由一个人完成，也逐渐变成了由很多不同角色的人共同合作来完成。

12.4 软件的作用

在软件出现之前，人们活在现实生活中，每个人干自己的工作，自行保存自己的工作结果。人们面对面或者通过电话等方式沟通，如下图所示。人的各种信息的传递都受到物理空间的限制，信息保留的时间有限，很快就会消失。



有了软件之后，软件工程师把人们日常生活中所做的事情，虚拟到计算机中。为了操控计算机中的虚拟人，人们需要把自己本人一起虚拟化到计算机中。而人们的日常活动方式则变成：通过计算机的输入输出设备来控制计算机中的自己，完成日常的工作以及与他人的沟通。如下图所示。



这样，空间和时间的限制就都被打破了，人们不再需要在现实的空间下活动。软件模拟出了一个虚拟的世界，人们在自己创造的虚拟世界中互相联系，形成了一个虚拟的社会，从而给人类带来了极大的便利。人们对世界的访问生命周期也发生了切分，形成了软件的访问生命周期，从而打破了物理时空的限制。对时空

限制的突破，极大地提高了生产力，人们不再需要花大量的精力来跨越时空的限制，只需用很短的时间就能够做到更多的事情，从而完成更多的生命周期。

模拟人类的行为始终是软件的一个目标，包括模拟人类的各种感觉以及人类大脑的计算。比如模拟人类社会（互联网社交），模拟交易，包括现在正在流行的VR、人工智能，等等。模拟的对象越来越高级，难度也越来越大。很多原来不可想象的事情，因为软件的发展，现在都变成了现实。

未来不管如何发展，模拟人的所有行为都是一个大的趋势。但是不管如何模拟，人类自己的核心生命周期：生、老、病、死、吃、喝、拉、撒，还是需要自己来做，无法由软件或其他的东西来代替。也就是说，软件的主要目的，就是把人类生活的非核心生命周期软件化、虚拟化，以提供更低的成本和更高效率的新生活，让核心生命周期的运行能够更加容易，让非核心生命周期的处理更少地占用人类的时间，变相地延长人类生命。从这个角度来看，软件主要依赖的还是人类的生活知识，而人类的生活知识又受限于大自然。正如老子的《道德经》中的那句话："人法地，地法天，天法道，道法自然"，我们可以在前面再补上一句："软件法人"，"人法地，地法天，天法道，道法自然"。

第 13 章　软件的生命周期

人类社会发展的动力，是人们希望能够在更短的时间内做更多的事情。通过把自己不擅长的事情切分出来，交给其他擅长的人来做，从而形成分工，增强自己的核心能力。软件出现之后，人们慢慢地开始把自己擅长的事情也交给软件来做，人们可以去做更有价值，或者学习自己不会的事情。毕竟重复自己已知的事情不会让自己有更大的提高。软件工程师就是专门训练计算机新技能的人。

软件作为新出现的事物，也有它自己的核心生命周期，也就是它自身的“生、住、异、灭”。一个软件，因为某个业务虚拟化的需要而产生；后续不断地更新、修改，推动软件逐渐变异、长大；当该软件不再被需要（因业务的变化），或有更好的软件来替代时，该软件就会被废弃，完成使命而消亡。

软件的整个生命周期也会发生切分，从而形成两个子生命周期：软件开发生命周期和软件运行生命周期。

这两个生命周期里面哪个才是真正的核心生命周期呢？软件开发生命周期的主体是软件开发，软件运行生命周期的主体是软件本身。所以软件运行生命周期才是核心生命周期，因为软件运行生命周期的主体和大的生命周期一致。

从另一个角度来说，人们真正需要的是软件启动后为我们带来的服务，也就是说软件运行生命周期才是人们真正需要软件的地方。

而软件开发生命周期是为了产生软件，是为软件运行生命周期服务的。围绕着软件的生命周期，还会切分出很多其他的非核心生命周期。主要有如下几个：

（1）软件的开发生命周期。该生命周期的目的是为了产生可运行的软件，是可以切分出来单独管理的，这也是为什么会出现很多的软件代工。内部还会发生切分，如需求生命周期、代码开发生命周期、测试生命周期等。

（2）软件的运行生命周期。软件第一次启动才是真正的出生，软件的运行是

我们真正需要的核心。软件运行过程中，不断地积累信息，逐渐地壮大，直到形成新一轮的重生。软件的运行会形成运行的生命周期，从启动到停止。在软件的生命周期中，可能会包含多个从启动到停止的生命周期。该生命周期又包含：

a. 软件的访问生命周期。软件运行期间被访问，每次访问也是一个生命周期。

b. 软件的功能生命周期。软件的内部功能被使用时，也是一个个的生命周期，这个生命周期可能会包含多个软件的访问生命周期。

c. 软件的监控生命周期。在对软件运行状况进行监控时，就会形成软件的监控生命周期。软件第一次启动时，监控生命周期也随着开始；软件废弃，监控生命周期也随之结束。

下面对不同的生命周期做一个大致的介绍。

13.1 软件的开发生命周期

软件不是随便就创建的。因为一旦创建，就必须要有资源维护。所以在软件开发组织的时候，是否需要创建一个软件，要由大家一起来讨论，认证其必要性，确定代价和所获得价值的对比，也就是 ROI （Return On Investment）。一旦确定需要创建，就需要安排一系列的资源来支撑这个软件的生存。比如研发人员、机器、监控等。所以从这个角度来说，能够在一台机器里面独立运行起来的才能算一个软件。

软件不是创建好就能够运行的，它需要用代码来编写，以实现对业务的模拟。代码确认无误后，再把代码编译成机器代码，从而形成一个部署单元，部署到机器上运行。此时软件才真正诞生，同时一个软件的开发生命周期就结束了。

从软件运行生命周期角度来说，一个可运行的独立部署单元才算是一个软件。

所以从决定开始生产一个软件，到软件真正地运行起来，会经历一段时间的代码积累。这个过程也是一个生命周期，叫作软件开发的生命周期。这个生命周期，以项目启动为始，然后和业务人员学习业务（也就是业务逻辑）形成需求；接着根据需求完成编码，并测试代码是否达到业务模拟效果；测试通过后，再经由业务人员确认，部署上线。上线成功后，一个软件的开发生命周期结束。

13.2　软件开发的增长

软件工程师是软件开发生命周期中的关键人物，软件开发过程中的所有活动都是为了帮助软件工程师能够把代码写对，把业务虚拟出来。软件工程师首先必须要理解人是怎么在日常生活中完成业务工作的，然后才能够把这些业务工作在计算机中用代码模拟出来。

就人类的学习过程来说，需要时间的积累。比如：马尔科姆·格拉德威尔（Malcolm Gladwell）在《异类》（*Outliers: The Story of Success*）[1]中提出的一万小时定律：要成为某个领域的专家，需要一万个小时的训练。按比例计算就是：如果每天工作八个小时，一周工作五天，那么成为一个领域的专家至少需要五年。

另一个学习的判断标准是：学习一个东西如果只是能听懂，还不算学会；如果自己能够按照所学的执行，也才学会了一半；如果能够把自己所学的东西用自己的语言表达出来，让别人听懂，这才算差不多学会了。

软件工程是一门综合学科，软件工程师的培养本身就需要学习大量的计算机语言和计算机知识，还有相关的数学、物理、电子电路等知识，门槛很高。

而要想对软件和计算机之外的行业业务实现模拟，软件工程师还要对该业务所在行业的专业知识进行一定的积累，并要超越听懂和能执行两个阶段，才能够用另外一种语言，也就是计算机语言表达出来。这是一个相当高的要求。

软件工程师一旦进入比较复杂的业务领域，往往就会力不从心，因为业务已经超出了自身的能力。因此，软件开发的瓶颈需要被打破，需要引入大量的开发人员，慢慢就开始有了分工。

按照前述的架构切分原则，软件开发要想通过分工达到增长，就需要把软件开发的生命周期识别出来，并进行切分。软件开发的核心生命周期是编写代码，这是软件运行生命周期启动的前提条件。软件工程师负责编写代码，完成编写代码的生命周期。在这个核心生命周期下，软件开发生命周期又被切割成不同的非核心子生命周期，每个非核心子生命周期围绕核心生命周期组成树状架构，在时间和空间上并行开展工作，以提升软件开发的产量和速度。比如，对于软件工程师不熟悉的行业知识，会剥离出来给熟悉业务的人员，如业务分析师（Business

1 《异类》，马尔科姆·格拉德威尔在 2008 年发表的著作。

Analyzer，BA），来进行行业知识和业务的识别；系统的设计会交给架构师来执行；设计的编码实现则交给开发人员；编码结果的检验交给测试人员。还有很多其他角色来配合他们的工作。比如为了组织这些角色协同工作，还需要有项目经理的参与。每个角色都有自己的生命周期，也有其独特的特质，这里不再一一展开。

无论是哪种切分或分工模式，即软件开发模式，其目标都是达到软件开发的增长。

所有达到增长的手段，都是围绕软件开发核心生命周期进行切分的。把可以并行的非核心生命周期切分出来，以提升不同人员工作的时间和空间的并行度。这就是软件开发的架构。不同的拆分方式，形成了不同的软件开发模式。

13.3 软件开发的迭代

软件上线后，还会不断地进行修改。有时是因为要增加新的需求，有时是因为软件本身存在一些问题，比如出现错误（Bug），需要修复。此时软件会重新再走一遍软件开发流程，这就是软件开发的迭代。

因为迭代的存在，软件会产生不同的版本。每个版本的软件都是一个完整的开发生命流程。软件的诞生可以认为是第一个版本。版本叠加的过程也就是软件不断长大的过程。

由于软件开发是一个独立的生命周期，可以与软件生命周期并行，因此自然就可以做到多个版本并行的开发，以并行的方式提高生产力。一旦同一个软件的不同版本生命周期并行地推进，就需要把代码的生命周期单独切分出来，以确保不同版本软件工程师之间的代码，在空间和时间上不会产生冲突。因为软件工程师编写代码是软件开发的核心生命周期，所以要确保他们的工作能够按照各自的时间顺序地执行编码生命周期活动。

多版本并发又会产生上线冲突的问题。线上版本的运行是软件生命周期的核心，不管有多少个开发生命周期并行，线上版本都只有一个，必须确保版本上线的顺序发生。另一方面，软件开发生命周期是软件生命周期中的非核心生命周期，不可以损害核心生命周期，所以必须要排队上线，并且要保证后续的版本要包含前一个线上版本的所有内容，确保其连续性。这就形成了发布生命周期，后续会

继续展开讨论。

13.4　软件的运行生命周期

软件上线之后，运维人员需要把软件在部署的机器上启动。如果是服务端软件，则需要服务提供者启动。如果是客户端软件，则需要客户自行启动。此时软件的核心生命周期就启动了，软件开始服务于它的用户，执行对业务的虚拟化，直到软件停止为止。

在这期间，每一个用户对这个软件的访问，都是一个访问生命周期。软件的虚拟化的成效就是靠访问来达成的，所以软件的访问生命周期很重要，展示了用户的使用情况。软件功能的使用情况，也会形成功能的访问生命周期，这个生命周期往往和业务的子生命周期相关。

在软件启动后，运维会对它进行健康监测。就好比监控一个人一样，看这个“人”是否还活着，活得是否健康。业务也会针对该软件做业务监控，看企业业务虚拟化的成效，了解用户的使用情况，以便为下一步做好准备。

这些新生成的非核心生命周期，都形成了树状结构，围绕在软件核心生命周期周围，共同组成了软件的生命周期。

第 14 章　什么是软件架构

前文探讨了什么是软件，以及软件的生命周期。在软件行业里，一谈架构很多人下意识地认为是指软件架构，甚至认为软件架构就是指软件的技术，或者是软件技术的更高体现形式。其实架构是一个更普遍的概念，软件架构是最近几十年随着软件的出现才兴起的概念。那么什么是软件架构呢？应用前述的办法，先尝试分析一下这是什么问题，是谁的问题？利用生命周期分析，找到核心生命周期，以及核心生命周期的主体。

14.1　要解决什么问题

软件的目的是把现实生活模拟到计算机中，并且软件是需要在计算机的硬件中运行起来的。要做到这一点需要解决两个问题。

（1）业务的问题：在现实生活状态下，没有软件的时候，解决问题的主体是谁，解决的是什么问题，又是如何解决、如何运作的？如果业务是模拟自然界，那么自然界又是如何运作的？需要明确业务的主体，并深入到业务的生命周期变化中，找到业务独特的个性。

（2）计算机的问题：软件的生命周期可以拆分为两个子生命周期，软件开发生命周期和软件运行生命周期。

a. 在软件开发生命周期中，如何用计算机语言来表达业务的生命周期？现实生活中业务生命周期是如何切分的？哪个是业务核心生命周期？哪些是业务非核心生命周期？业务生命周期拆分树是如何形成的？明确了这些问题的答案之后，就可以明确软件的拆分，以及软件的内部组织。

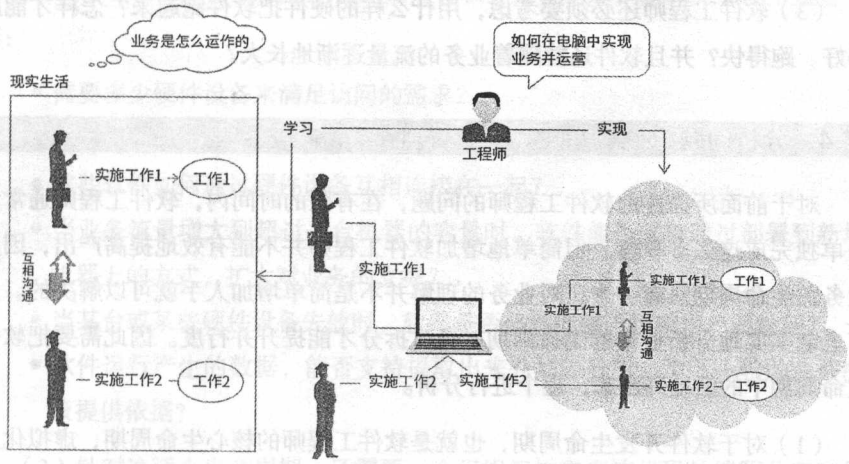
b. 在软件运行生命周期中，模拟业务的软件需要哪些硬件设施才能够满足要求？并且当访问量越来越大的时候，软件能否支持硬件的逐渐长大和性能线性扩展？

- 硬件是有可能会失效的，软件如何在硬件失效的情况下，仍然能够保证可用性，让用户能够不中断地访问软件提供的服务？
- 如何收集软件产生的数据，为下一阶段的工作提供依据？

14.2　分别是谁的问题呢

（1）业务的问题：业务的所有人（Owner）需要提升业务的效率，通过更多的虚拟人代替现实的人，提升生产力，降低业务的成本，这是动机。业务的所有人就是该问题的主体，软件开发的出发点就在这里，推动力也在这里。

（2）计算机的问题：软件生命周期负责的主体是软件工程师，因而要解决业务所有人把业务虚拟化的问题，并且要解决软件开发和运行的生命周期的问题。如下图所示。



14.3　分别有什么问题

（1）业务问题的本质是业务所服务对象的利益问题。根据业务对象的利益，可以整理出业务的生命周期和生命周期的拆分，再根据业务的概念和组织方式，可以分析出业务的核心生命周期。根据当前业务增长的方式，也可以反过来理解业务的生命周期拆分。通过对生命周期的分析，可以快速地理解业务，并进行领域建模，为软件模拟业务做好准备。

（2）软件工程师的问题本质，就是要用代码将业务模拟出来，形成软件，交

给计算机运行。为了能够让业务在软件中很好地跑起来，软件工程师必须理解业务的生命周期、拆分方式，以及业务服务对象的利益所在，即业务问题。业务的核心生命周期是什么？面对这些问题是如何拆分解决的？涉及了哪些概念？ 这些概念分别解决了哪些问题？针对这些问题，软件工程师经常根据自己的理解，创建一套概念体系来表述业务，或者用软件行业的术语来表述业务。如果这么做，会导致以下两个问题：

- 业务人员无法和软件工程师交流，因为他们无法理解软件工程师所创造的概念或者软件行业术语，也无法确认软件工程师的理解是否正确。
- 软件工程师所表述的东西，并没有在实际业务中实践过，业务人员也不知道这些概念是否能够解决业务的问题。如此一来，就很可能出现不仅没有解决问题，还创造出了更多的问题。

（3）软件工程师还必须要考虑，用什么样的硬件把软件跑起来？怎样才能跑得好、跑得快？并且软件还能随着业务的流量逐渐地长大？

14.4 分析问题

对于前面所描述的软件工程师的问题，在有限的时间内，软件工程师通常无法单独完成这么多事情。但简单地增加软件工程师并不能有效地提高产出，因为业务的生命周期只有一个，对业务的理解并不是简单增加人手就可以解决的。只有老老实实地分析业务的生命周期，通过拆分才能提升并行度。因此需要把软件生命周期中的活动列出来，逐个进行分析。

（1）对于软件开发生命周期，也就是软件工程师的核心生命周期，虚拟化业务需要完成下面这些事情：

- 学习业务知识，认识业务的生命周期，以及生命周期中所涉及的利益相关人（Stakeholder）的核心利益述求。业务是如何拆分生命周期的，拆分出来的核心生命周期是哪个？这些生命周期是如何组织起来，并通过怎样的组织架构完成整个组织的核心利益的？业务的运作流程，也就是核心生命周期的运转，涉及哪些概念，有哪些权力和责任等。
- 通过对业务知识的学习，确定核心生命周期和非核心生命周期，以及这些生命周期的组织方式。针对这些生命周期所对应的权力和责任以及组织架

构，对业务进行建模，并把建模的结果用编程语言实现。这就是业务的运作模型，也就是虚拟人的组织架构，对应的是现实生活中的组织架构。现实生活是软件思想的最主要来源。

- 理解参与业务的利益相关人是如何和业务打交道，并为每个角色的权力和义务进行代码描述并落地实现的。通过编程语言，利用业务模型的能力，围绕核心生命周期把不同的生命周期合理地组合起来，实现用户访问业务生命周期的通道。这部分变化是最频繁的，因为直接面对业务的操作需求。
- 考虑如何把业务运行的结果持久化，并通过合适的手段把持久化后的数据，在合适的时间合适的地点加载出来。这部分和基础设施有关，变化也会比较频繁，相当于人类把记忆存储在外部的设备中，如文件柜，并按需快速加载。

（2）对于软件运行生命周期，即如何让软件运行起来，需要完成以下这些工作：

- 需要多少硬件设备来满足访问的需求？
- 软件要如何拆分并部署到哪些硬件设备上？
- 这些软件如何通过硬件设备互相连接在一起？
- 当业务流量增大到超过一台机器的容量时，软件能否支持通过部署到新增机器上的方式，扩大对业务的支撑？
- 当某台或某些硬件设备失效时，软件是否仍然能够不影响用户的访问？
- 软件运行产生的数据，能否支持提取出来并加以分析，为下一轮的业务决策提供依据？

（3）针对这两个生命周期，还需要一个组织架构来实施代码的编写分工和软件运营的分工，需要确定的问题如下：

- 要想完成软件开发生命周期和软件运行生命周期中所列出的这些事情，需要哪些角色参与？
- 这些事情基本都需要顺序地发生，如何保证信息在不同角色的传递过程中不会有损失？ 或者即使有损失，也能快速纠正？
- 这些角色之间如何协调才能共同完成虚拟化业务的需求？

14.5 会生成哪些架构

如果业务足够简单，用户流量足够小，时间要求也不急迫，那么一个人，一台机器慢慢实现就可以了。这种情况一般不需要讨论架构的问题，因为不需要并行，即没有时间压力，也没有产出的压力，事情一件一件去做就好了。此时对应的业务通常比较简单，对应的业务组织架构也会比较简单。

当软件对应的业务组织架构越来越复杂，或者访问的流量越来越大时，就会产生时间的压力，即在相同的时间内要完成更多的产出。软件会拆分得越来越复杂，机器也会越来越多，同样也需要越来越多的人编写拆分出来的软件，甚至同一个软件也需要拆分为多个组件，需要多人合作完成编写。但是业务的核心生命周期，即业务的核心建模不会有任何的变化，还是完成同样的这些事情。唯一的区别就是会拆分出很多的非核心生命周期，它们以核心生命周期为根，不断地增长，最终形成一个树状架构。生成的架构如下：

（1）在软件开发生命周期中参与的人员越来越多，就会形成软件开发团队的组织架构。因为软件代码开发的过程是一个生命周期，严格按照时间有序推进，会根据开发的核心生命周期进行拆分，形成很多个非核心生命周期。按照软件开发核心生命周期的流程，把不同的非核心生命周期串联起来，使得这些非核心生命周期能够并行开展工作，这就是软件工程。所形成的就是软件开发团队的组织架构。

a. 为了让业务在软件中实现并落地，并便于用户对软件的访问，需要前端人员、业务代码人员、存储人员等不同技巧的人同时工作，并把代码进行切分，形成代码的架构。切分的原则就是根据用户对软件访问的生命周期，识别出核心生命周期和非核心生命周期，形成树状架构。这就是为什么会形成代码的分层。

从分层的角度进行思考的时候，要千万注意不要破坏树的架构。有树才会有分层，有分层却不一定有树。

当这些工作由一个人来完成的时候，由于没有明确的分工，不一定会有代码架构，所以代码往往会比较混乱。

b. 为了完成业务的工作，需要把识别出来的业务架构和支撑业务的组织架构，以及业务运作的核心生命周期流程，用代码表述出来。这时被虚拟化的业务架构

和组织架构，也需要体现在代码中，保持和现实生活中一致，形成虚拟人的组织架构。这就是业务模型的架构，也是代码架构中的一部分。

（2）在软件运行生命周期中，当软件的流量越来越大时，慢慢就会超出软件所在机器的能力。这个时候就必须增加机器，软件所部署的机器也会根据用户访问的生命周期，按照树状的结构开始拆分，所形成的是硬件部署树状架构和软件部署树状架构。这也是为什么会形成部署的分层。

每台机器都是有生命周期的，每个软件也是有生命周期的。针对机器和软件的生命周期监控，本身也是一个生命周期，即监控生命周期。软件的监控生命周期需要管理起来的，所以同样需要一个人员的组织架构来保证软件监控生命周期的推进，即软件运维监控团队。

14.6　什么是软件架构

通过上文的分析可以看到，软件架构的内涵比较多。总而言之，软件架构就是通过对软件生命周期的拆分，在符合业务架构的前提下，以达到软件本身访问增长目的的方式。这个增长需要软件开发的增长，也需要软件运行的增长，由此达到所支撑业务的增长。

软件架构离不开软件开发团队的组织架构，这个组织架构是软件开发生命周期和软件运行生命周期的执行者。

离开了组织架构，任何软件架构设计都是纸上谈兵，因为架构的核心生命周期就是架构的执行。

很多人认为架构是进化或演化出来的。进化的含义是由一个物种变成了另外一个物种，是本质的变化。比如物种由水生进化到了陆生。架构追求的实际上是业务不断的长大：通过对业务生命周期的拆分，突出并精简业务核心生命周期，拆分出非核心生命周期，达到不同生命周期在空间和时间上并行，便于不同的人同时开展工作，提升业务人员单位时间内产出的办法。

业务相当于基因，而架构树状拆分则相当于细胞的分裂。就好比每一个人，都是从一个细胞逐渐分裂出来的一棵树，起决定作用的不是分裂本身，而是他的基因。基因决定了细胞最终会分裂生长成什么样的一个生命。比如一棵树从种子

长成小树苗，再长成参天大树，这不叫进化，这叫长大。长成什么树是由树的基因决定的，不是架构。因为不管怎么拆分，业务的目标，也就是基因没有任何变化。如果一个企业的业务由制造商变成电商，这个时候业务就发生进化了。因为基因变了，业务已经完全不一样了，整棵架构树的含义就变了。该企业的软件架构就需要重新设计，而不能在原有的架构上修改，也就不存在架构的进化。

而对于树的长大来说，架构进行的始终是树的拆分，把非核心生命周期从业务生命周期中拆分出来，达到空间或时间上的并行。所以严格来说，只有业务才会进化。架构是支撑业务长大的，形成的是新的拆分。业务的核心生命周期，也就是架构树的主干，并没有变化，即没有变成一棵不同品种的树，因此不能叫作进化，只能叫作架构树上新的架构拆分。随着架构拆分得越来越细，树长得越来越大，并行度越高，业务也就长得越大。

从另一个方面来讲，当原有的结构不是树状架构时，需要把它转变为树状架构，或者由一种结构转变成另外一种结构，似乎就是所谓的架构演化想表达的意思。可是如果该结构的拆分不是以增长为目的，则很难认为是架构，自然也无法称之为架构的演化，或许可以称之为结构的演化。当然，非要称之为架构的演化也并无不可，理解概念背后所解决的问题才是最重要的。

有人会问，为什么没有提到类似应用架构、硬件架构、数据架构、系统架构、基础架构等架构呢？这些名词有些在软件开发架构下和代码有关，有些在软件运行架构下和部署有关。必需从软件的生命周期一步一步切分下去，才能清楚地知道某个架构出于什么位置。具体内容，留待后续再展开讨论。

第 15 章　什么是软件架构师

在之前的文章中，已经尝试定义了什么叫作架构和架构师。其中架构师的主要职责是通过架构拆分达到业务增长，并根据架构拆分的结果给其他人分工，为增长结果负责的这样一个角色。其本身也是一个分工，这个分工往往决定了架构师在组织架构上会是一个领导者的角色。在软件行业里，我们经常会看到软件架构师这样一个职位，拥有这些职位的人都有架构师的头衔，比如应用架构师、系统架构师，等等。下面试着讨论一下软件架构师的不同之处。

15.1　软件架构师的区别

在大部分行业中，架构师做的工作大都是本行业的内容，和所做的业务直接相关，对于所做的业务都是知根知底的。而在软件行业则有很大的不同，软件主要是给其他行业服务的。从前面所给的软件定义也可以看出，软件的核心是模拟人类的业务，而不在软件本身。而从软件人才的培养模式则可以看到，软件从业人员培养时需要学习的主要是计算机、软件相关的知识，是为计算机和软件这个行业服务的。现实和目标会产生一个比较大的差距。

这一点其实和语言类的行业有点像。很多人学习英语等外国语言，但是最终从事语言本身研究的人寥寥无几，大多数人往往是服务于其他的各行各业，需要重新学习所服务行业的行业知识。所以这个问题也不是软件行业所独有的。

在现实生活中，具备软件架构师头衔的人，大都是从软件工程师中成长出来的。这个出身已经决定了他们的主要关注点是在软件和计算机技术上，并不会很关心业务本身。并且这部分人往往都有一个主流观点：“不写代码就没有资格叫架构师”。这些软件架构师其实就是软件技术能力比较好的软件工程师，并不一定是我们前面所定义的软件架构师。

具备架构师头衔的人并不一定是架构师。

15.2 软件架构师的困境

对于从软件工程师成长出来的软件架构师，要真正成长为一个架构师，首先需要克服的就是时间困境。从软件工程师的角度出发，首先需要面对的就是用自己的软件技术能力去解决业务问题。这些业务对于软件工程师来说，是另一个行业的内容，是“别人”的问题。因此软件工程师对于业务往往一知半解，也没有太大的动力去研究，他们主要专注于完成自己手头的编码工作。如果一个人在工作中，只是致力于完成自己的工作，以做好自己的工作为主要目标，那么用自己行业的知识去理解另一个行业，就很容易出现沟通的困境，造成很多理解上的困难。前面“削土豆”的笑话只是冰山一角。

当软件工程师需要帮助别人解决问题，并且按时、按需解决业务问题已经成为他们自己的问题的时候，软件工程师就有了时间的压力，潜意识里会自然而然地产生对时间的恐惧。在潜意识里，这个恐惧会想方设法地推动软件工程师采用各种手段，如加班加点，以便及时地完成工作，换取报酬。要想成为架构师，则必须超越对时间的恐惧，看清楚需要解决问题的主体是业务人员，而不是自己，即需要解决的问题是另一个行业的问题，自己是在帮助业务人员解决问题。这也是软件特别的地方。如果只顾着完成自己的工作，而业务的问题没有解决，那么很可能需要返工，从而导致软件工程师投入更多的时间。要知道，工作是否完成由业务人员决定，而不是软件工程师自己。

为什么软件工程师会对时间有恐惧和压力呢？其原因是他们把按时完成自己的工作当成了自己的最大利益。人对时间的压力是与生俱来的，并且对业务的不了解也会导致他们没有太大的把握。这一问题在其他行业的表现并不明显，毕竟在其他行业，架构师主要处理的是本行业的问题，对业务比较熟悉。软件行业则较为特殊，通常是以业务的问题是否解决为判断标准，是在解决另一个行业的问题，这一点提高了对软件架构师的要求。这就要求软件架构师把完成业务的工作当成自己的最大利益，深入到业务中去。随着对业务的熟悉，对时间的恐惧才会慢慢地消失。对业务领域理解得越深入，就越知道如何去发现问题，慢慢就成为业务专家了。只有做到这一点，才能在业务领域建立自信，成为一个合格的软件架构师。

15.3　生命周期的思考

作为软件架构师，必须时刻把对软件的生命周期和业务的生命周期的识别放在第一位。软件生命周期的核心在于软件运行生命周期，以及围绕软件运行生命周期的拆分和组织，业务生命周期的核心在于围绕业务核心生命周期的拆分和组织。

对于软件生命周期，必须要深入思考软件开发的生命周期和软件运行的生命周期。在这个基础上，要根据业务的情况合理地进行软件开发生命周期的架构分拆，以及软件运行生命周期的架构拆分。合理的标准，要结合业务的流量做判断。软件开发的拆分和软件运行的拆分的目标都是为了支持业务流量的增长。

在代码层面，要对业务代码和访问代码做好架构拆分。业务代码是软件访问生命周期的核心，软件运行的拆分受限于软件代码的拆分。因此要确保业务代码符合业务的生命周期，使得业务生命周期活动的结果积累在生命周期的主体上，也就是内聚性，避免散落到访问代码中。这样软件的拆分就不会有太大的问题。

软件的运行生命周期决定了软件的运维。对软件运行生命周期的拆分，就形成了软件的运维体系。

这些生命周期都是软件架构师必须要深入思考的，以确保架构拆分之路畅通。

15.4　软件架构师的权力

软件架构师需要去拆分生命周期，并要形成组织架构去落实架构执行，而且要平衡别人的利益，甚至去调整别人的利益。软件本身包含软件和业务两部分。对于软件架构师而言，是无法直接去调整业务利益的，只能够某种程度地去影响。毕竟软件架构师是为业务服务的，而不是主导业务的业务架构师。但对于软件的开发生命周期和软件的运行生命周期，软件架构师必须要具备权力去调整。这一点要求软件架构师必须是一个软件团队组织领导者的身份。

在软件生命周期拆分树中，软件架构师本身的业务也会发生架构拆分，拆分出不同位置的架构师就有了各自不同的架构范围：有负责部署架构的架构师，如系统架构师；有负责数据架构分拆的架构师，如数据架构师；有负责应用代码编写架构的架构师，如应用架构师，等等。还有很多其他不同类型的软件架构师，种类多少取决于架构师业务分工的细化程度，这里不再一一列举。

在软件行业，很多公司设了软件架构师的职位，主要职责是做出架构设计，也具备一定的影响力，但并不具备调动组织架构的权力。这样的职位往往达不到架构师的效果，有时候还会起反作用。因为架构师只能够通过建立某些流程来行使架构师的权力，比如强制架构 Review，给出架构建议等。时间一长反而会造成很多不必要的内部冲突，最终会导致沟通成本增加，减慢研发的速度，同时也会导致这些流程流于形式，大家敷衍了事，反而增加了研发成本，得不偿失。比如有些团队为应付架构 Review，会做两套架构，一套用于 Review，一套用于实施。相信很多人都经历过这些，但似乎很少有人探讨这是为什么。

架构师的核心在于架构的执行，软件架构师必须是一个组织的领导人，有权力调动这个组织的架构，才能够更好地发挥架构师的作用，才能够把软件开发生命周期、软件运行生命周期和业务生命周期的拆分落实执行。

软件开发团队的组织领导人其实都是架构师，只是没有这个头衔而已，真正的架构师不一定具备架构师的头衔。

一个好的领导，就是一个很好的架构师。在这个架构师的领导下，这个组织一定是健康向上的。因为好的架构师会关注核心生命周期，能够通过合理地拆分保证权力和义务的对等。对等就意味着权力得到了真正的下放，各角色的激励也能够到位。并且这类领导对业务的组织架构变化也会很敏感，会及时地根据业务组织架构的变化，根据软件开发生命周期和软件运行生命周期进行对应的组织架构调整，在问题发生之前就把问题消弥于无形。“无为之治”似乎就是这个意思。这个组织会具备更好的抗压能力，能够更好地为业务服务。这个组织的成员内心一定都是比较平衡的，每个人的能力都能够得到比较好的拓展。由此也可以看出：

人类架构的核心就是组织架构，正确的组织架构才能保证架构的执行。

15.5 软件架构师和技术人员对技术的态度区别

前面提到：“不写代码就没有资格叫架构师”，这是软件行业流传比较广的一句话。同样还有另外一句，“不懂技术的无法当架构师”。很多人拿写代码多少和懂得技术的多少来举例，认为这样才是真正的架构师。这是从软件工程师成长起来的人的普遍看法。软件特别的地方在于解决业务的问题，所以也会有一些觉悟比较高的人说：“不懂业务的架构师不是一个好架构师”。这个提法更接近软件的

定位。说明软件行业也在慢慢地理性化，这是好事。

从技术的角度去看软件架构师，就像雾里看花，是无法真正理解软件架构师的。大自然是最好的架构师，但大自然并不懂得如何去实现每个生命、每种技术，她所做的只是让每种生命能够顺应自己的生命周期而已。

要想做好架构师要工作，就要向大自然学习，这样才能够认识到事物自身的生命周期，并能够去顺应事物自身的生命周期的规律来进行拆分，以达到增长的目的。

架构师并不排斥技术，但架构师和技术人员对技术的态度是不同的。

- 技术人员对待不同技术是有区别的。往往是什么技术流行就追什么技术，恨不得在自己的项目上把所有的新技术都用一遍。因此，技术人员也热衷于创造新的技术，以对技术的掌握作为自己的标签。并且技术人员对于技术是有感情的，会喜欢或者热衷于某些技术，而对于其他的某些技术则嗤之以鼻。也就是说，技术人员对于技术是不平等的，这个不平等，既会导致不同技术的诞生，也会导致某些技术走向极致。这些都不是坏事，这是技术人员的特性，有利于技术的进步。另一方面，追求新技术会导致技术人员被技术所困，疲于奔命。虽然如此，仍有很多人乐此不疲。
- 而架构师则不一样，架构师很冷静、很平等地对待所有的技术。对于软件架构师来说，他们要深刻地领会软件的开发生命周期和软件的运行生命周期，以及业务的生命周期，把它们合理地结合在一起，并能够通过软件的快速增长来支撑和顺应业务的增长。这些是软件架构师的核心能力。为达到这些目标，无论是新的还是旧的技术、先进的还是落后的技术，只要场景合适，他们就会采纳。因为架构师是技术的使用者，他们把技术当作解决增长问题的手段和工具，而不会被技术束缚住。

15.6　架构师是技术的使用者

想做好技术的使用者并不容易。架构师必须深入地了解不同的技术，知道这些技术的强项和弱点，懂得合适的取舍。比如作为一个建筑架构师，建造砖瓦结构的房屋，如果不了解砖、瓦是怎么制作出来的，就无法理解砖、瓦的特性、弱点以及寿命，也就使用不好砖、瓦。如果没见过砖瓦匠如何使用砖瓦砌墙的，就

不知道有些砌法是中空的，有些是实的。不同的砌墙技术，其承重能力如何，保温性能如何，等等，都无从得知。这样是无法使用好技术的。同样，无论是采用钢结构，或是采用木结构，都要对这些材料，以及现存的技术有一个比较通透的了解，才能够做出靠谱的判断，但并不是说架构师必须是所有技术的专家。比如木结构里面的“榫卯”[1]结构，架构师可以很了解并使用。但让架构师自己来做一个“榫卯”结构，则并不容易，木工活要非常精细才行。当架构师需要一个“榫卯”的时候，还是要找木工来解决问题，而不是花时间去学习木工，自己来做。

技术本身是没有好坏的。比如最早的计算机软件是单机运行的，逐渐发生了架构拆分，形成了 C/S 的架构，也就是 Client/Server，随着互联网技术的发展，变成了 B/S，也就是 Browser/Server。其实是换汤不换药，Browser 还是一个 Client。但是因为 Browser 的普适性，其实就是又做了一个架构拆分，成了一个通用的 Client，做到了很多 C/S 时代做不到的事情。互联网时代的技术人员对 C/S 架构鄙视得一塌糊涂。到了 App 时代，又回到了 C/S 时代的做法，因为 HTML5 响应慢，无法适应手机客户端。随着手机端带宽的逐步改善，以及手机端计算能力的逐步提升，慢慢可能又会走向 B/S 模式。所以说，随着人们问题的不断变化，会有不同的技术周期出现，技术有自己的生命周期。所谓“三十年河东，三十年河西”，没有永远最好的技术，也没有永远最差的技术，而问题总是在不断发生变化的。对于这一点，架构师要有清醒的认识。

15.7 如何保障架构落地

使用一个技术就好比是搭积木，只要问题明确，寻找技术并组合来实现架构的拆分并不是难事。如果架构师能够深入理解技术背后的驱动力，了解技术是如何实现的，则能够做出更合适的判断和选择，这就是所谓的“接地气”。所以架构师在做好架构拆分后，会形成不同的生命周期。针对不同的生命周期，要选择不同的技术来进行支撑，再把不同的生命周期分工交给不同类型的软件工程师去实现。这就是架构师和技术人员的分工配合：架构师拆分生命周期，技术人员实现生命周期。

---

1 榫卯（sǔn mǎo），是古代中国建筑、家具及其他器械的主要结构方式，是在两个构件上采用凹凸部位相结合的一种连接方式。

这就是为什么架构师需要有组织架构的权力，因为要确保架构拆分的落地。技术更多的是需要软件工程师掌握的。架构师思考技术时则更多地考虑技术对生命周期拆分的支撑，以及不同技术实现拆分时落地的成本和收益。以最少的成本达到更高的增长，是每个架构师追求的目标。所以架构师既不是技术的对立面，也不是技术的同一面，两者是相辅相成的关系，毕竟都是从软件工程师这个角色中拆分出来的。

如果把业务认为是硬币，则可以认为架构和技术是硬币的两面，架构和技术共同形成了业务。

技术是架构师手中的工具，当没有合适的技术时，架构师会去创造技术，或者催生出新的技术。比如架构拆分往往会导致新的生命周期被剥离出来，新的生命周期和原有生命周期的组合也会产生新的问题，这会迫使技术人员来解决这些问题，催生出新的技术。

如果软件架构师不具备组织架构上的权力，那么架构师的设计慢慢就会被架空，无法落地。而软件工程师则会按照自己的想法自行其是，往往不计代价地采用流行的技术或自己喜欢的技术，甚至会和架构师产生冲突。或者软件架构师逐渐变成软件工程师，慢慢专注于实现自己的架构去，而软件和业务的增长也就无暇顾及了。

其实对于做软件的技术人员而言，技术也可以分为两部分：一部分是软件技术，另一部分是业务技术。当前软件行业所说的技术，基本上都是指软件技术和计算机相关的技术，也就是软件的访问生命周期所涉及的技术，比如服务、存储等相关的技术，或计算机硬件设备相关的技术。软件技术大部分集中在软件的访问生命周期部分。业务技术这一块，软件工程师则较少涉及，这部分主要隐藏在业务逻辑中。因此在谈技术时，只谈软件技术是远远不够的。而不管是软件技术还是业务技术，均来源于对现实生活问题的解决，现实生活才是架构师真正的养分来源。想通这一点，离架构师就不远了。

技术人员如果要成为架构师，就必须跳出技术的视角，换一个角度去看技术。要把时间花在研究生命周期规律和业务的增长上，花在选择合适的技术上，而不只是追求新潮的或自己喜欢的技术。

软件行业为何需要一个架构师的职位？大部分原因在于软件开发组织的领导

者太看重技术，也有可能是因为这部分的领导人，大多是从软件工程师成长起来的、从技术领域成长起来的。具备架构师头衔但没有执行权力的架构师职位更像是一个“军师”，给组织领导人一定的架构建议以供参考，却不必承担任何责任，当然也不会有太多的权力，所以这个职位不是真正的架构师。而组织领导者应该回归到架构师的本位，虽然没有架构师的头衔，但实际上做的却是真正架构师所做的事情。

第 16 章　业务、架构和技术三者的关系

某天和朋友吃饭正好聊到这个话题，作为架构师或者做技术的人，在开发软件时，他们基本就是在扮演上帝的角色：不但要创建出一个个的程序，还要让这些程序能够脱离他们在硬件上独立运行，以便为这个程序所服务的群体提供服务。当这个程序出现问题或者错误（Bug）的时候，他们还要扮演牧师的角色，去修复这些问题。这不正是一个程序的社会吗？ 和人类社会的演变何其相似！软件是模仿人类的，用人类演变的历史来指导软件开发工作是一个很自然的想法，毕竟再次经历人类演变发展的历史是很痛苦的。架构师和程序员是决定软件的关键人物，可见架构师和程序员在扮演着多么重要的角色。

在软件设计开发的过程中经常会看到，很多所谓的架构讨论实际上只是在讨论某些技术的技术讨论。在很多人看来，特别是软件工程师，架构和技术实际上是等同的。多学会了几种技术，就觉得可以做架构师了。或者学会的技术越多，就觉得自己的架构水平越高。对于别的架构师当然也采用同样的标准来评判。要知道，任何技术都是为了解决某种问题而存在的，学会了很多技术，并不代表能够利用这些技术来解决问题。学会的技术的多少，所带来的差别只是解决问题的手段多了些而已。但是手段多了就一定是好事吗？ 学会的技术越多，很多时候越不知道采用哪种技术更好，所谓“乱花渐欲迷人眼"。

还有另一种很普遍的现象：做技术的软件工程师往往看不上业务。觉得技术更高端，而业务太平凡、太低端，并且业务人员总是给技术挖坑。而业务人员则觉得做技术的眼光高，但总是理解有偏差。技术人员往往对业务一知半解，业务问题总是解决得不圆满。但业务人员对此又无可奈何，因为自己不懂技术。

业务、架构和技术三者都是软件行业从业者必须打交道的，这三个概念到底有什么异同？大家应该怎么处理业务人员、技术人员还有架构师的关系呢？

16.1 什么是技术

当我们一无所有，或者什么都不会的时候，是谈不上什么技术的。就好比人类在最早期，什么都得用自己的双手来干。人们一旦在日常生活中无意间发现某些规律的时候，就可以通过人为地创造条件，让这个规律重复地发生。

通过人为创造条件，让指定的规律按照人类的意愿发生，这就是技术。

也就是说，技术的背后就是自然规律。技术出现后就会形成相应的工具，把技术实体化，以方便人们应用技术。比如取火，最早人类只能靠打雷等自然现象产生火。取火实际上就是人类的一个业务目标。

所谓业务，就是要解决人类的问题，目的是为了支撑人类自身的生命周期，使人类获得利益。

在没有取火技术的时候，人类只能靠不断地增加木材来保持火不熄灭。后来人类发现了钻木取火：只要用一个干的木棍，在另一个干木表面快速地转动，就可以产生火。这个办法让人类可以自行创造火源，这就产生了钻木取火的技术。

但是双手快速转动木棍钻木取火，需要很大的力量和速度，对人的要求很高，并不是所有人都能够做得到的。为了解决快速转动的问题，就有人采用弓弦来提升木棍转动的速度，可以让身体素质比较差的人也可以比较容易地生火。为什么加入弦就容易了呢？用双手转动木棍时，双手都要做同样的事：保持向下的压力，压紧木棍并搓动。也就是说，两只手同时做了两件同样的事，对每只手的要求都提高了，导致对人的门槛也就提高了。而引入了弓弦之后，一只手变成了只保持向下的压力，另一只手则只负责转动。引入弓弦这个技术的结果是对两只手的作用形成了分工，每只手只做一件事情，从而提高了每只手的效率，使得整个双手系统的能力和效率都得到了提升，钻木取火的效率自然就高了。也就是说：

（1）业务目标是为了取火，钻木取火这个技术的出现解决了这个问题。

（2）钻木取火的效率不高，影响了业务（取火）的效率，就有了进一步改进的动机，即对钻木取火的转动生命周期进行拆分，改进了转动木棍的方式，产生了弓弦转动木棍的技术。

（3）弓弦转动木棍的技术引入之后，导致了两手的分工，产生了架构。

16.2　业务和架构及技术之间的关系

技术总是在人类对业务目标要求不断提高的情况下产生，其目的是为了获取更大的利益。所以：

（1）技术是为了解决业务问题而产生的，没有了业务，技术也就没有了存在的前提。

（2）有了更好的技术后，效率较差的技术，就会慢慢地被淘汰，从而消失，一切都遵从人类的利益诉求。

有人会问，不用钻木取火了，但是弓弦加速转动木棍还可以用啊？是的，因为弓弦转动木棍这个技术，不是用来生火的，而是用来加速木棍转动的，所解决的问题并不一样。这个技术会更加广泛地用到其他需要加速转动的场景下，不仅仅适用于钻木取火，也不依赖于钻木取火而存在，而是变成了一个更通用的服务。

用生命周期来分析会更加清晰。物理课我们学过，摩擦生热的前提是有压力。钻木取火的核心生命周期是转动和压力两者的结合，核心是压力。钻木取火的生命周期是通过累积一个个摩擦生命周期产生的热，超过易燃物的燃点从而产生火，以火的产生为结束。在生命周期没有拆分之前，两只手都需要产生压力和转动。当把产生转动从钻木取火的生命周期中拆分出来之后，产生转动就变成了非核心生命周期，从而形成了一种用途更为广泛的服务。而钻木取火的生命周期并没有变，但是核心生命周期更加精简了，转动和产生压力变成两只手分别来进行，每只手只做一件事，并行执行的效率更高。甚至可以一个人负责产生压力，一个人负责产生转动，变成两个人的分工。两种不同的技术通过合理的架构组合起来，产生火的门槛就下降了，可以更有效率地解决业务问题。

因此不同技术之间有两种关系：

- 在解决同一个业务问题的前提下，更高效、更低成本的技术，会淘汰低效、高成本的技术。这是人类利益诉求所决定的。
- 通常刚开始解决核心业务问题的核心技术（钻木取火）的效率是比较低的，只是把不可能变成了可能。从这一点上来说，技术才是业务的使能者（Enabler）。慢慢就会有提高效率的需求出现，改进技术的要求就会变得很迫切。技术所解决的业务生命周期慢慢就会开始发生拆分。非核心生命周

期分离出去后，要么使用现有的技术来实现，要么形成新的技术，服务于更广泛的业务。

在业务生命周期拆分之前，技术是由一个主体串行地执行并工作的，这就是导致效率低下的原因。从业务生命周期中把非核心生命周期分离出去之后，非核心生命周期会形成新的技术，由不同的主体来执行。新的技术可以独立地与核心技术并行工作，提高了原有技术的效率。因为要解决的核心业务问题（生火）并没有发生改变，所以拆分所形成的是一个树状的架构。

也就是说，先有业务问题，才会有技术来解决业务问题。而业务的长大要求，提高了对技术的要求，导致了对业务生命周期的拆分，以并行的方式提升效率，形成了架构，也形成了新的技术。所以在这三者的关系里：

业务是核心，技术是解决业务问题的工具，而架构是让业务长大的组织方法。

架构需要用技术来实现拆分，而技术需要架构来合理组织，以提升效率。

这就是三者之间的关系。形象一点说，业务是硬币，架构和技术是硬币的两面。

一体两面，就是这三者之间的关系。

还要注意到，拆分出来的非核心生命周期的业务目标已经发生了改变，所形成的技术（弓弦转木棍）不仅可以用于生火，还可以服务于更广泛的业务，也不再依赖于原有的技术（钻木取火）而存在。所以架构拆分往往会催生出新的更普遍的技术。

16.3 技术人员和业务人员的关系

从上面的分析来看，业务、架构和技术之间是共生的关系，而不是互斥的关系。可是为什么技术人员总是和业务人员发生冲突呢？因为技术人员很多时候所关心的技术，和业务的主要目标往往不是直接对应的，而是软件架构树的分支节点。业务是负责某一部分的业务，也不和业务的主要目标直接对应，只是业务架构树的分支节点。只有直接解决业务问题的那个技术或业务，也就是树的根节点，才会和业务直接相关。一旦产生冲突，一般都需两个根节点（通常是领导），沟通才能解决问题。产生冲突往往都是因为业务团队和技术团队的组织架构不匹配造

成的，只有从组织架构上沟通才能解决。所以说，业务人员和技术人员的组织架构相互匹配、对应十分重要，可以减少大部分的沟通问题。

因为组织架构是树状的，所以上层节点都喜欢下军令状，要求下层执行，这也说明了执行才是架构的核心。同时权力也要下放，只有权力下放了，下层节点才能够执行好。

在软件行业，技术的根节点就是软件。因此架构师要认识什么是软件、什么是软件的生命周期以及什么是业务的生命周期。软件生命周期是如何拆分的、业务生命周期又是如何拆分的？理解这些问题才能够更好地组合不同的技术，完成业务的目标。

*软件里面和业务直接相关的，只有业务模型这一部分。用人来打比方，业务*模型相当于人的大脑，而软件整合业务模型采用的技术，全部都是计算机自己领域的技术，都是为了让用户对软件的访问可以到达业务模型，相当于访问大脑的通道。大部分软件工程师主要专注于访问通道部分，而真正应该投入的是大脑部分。很多架构师、技术人员主要专注于计算机相关的技术，看不上业务，背后的主要原因是对业务恐惧，更愿意专心于自己擅长的软件技术，因而忽略了业务，这也是为什么技术总是和业务相冲突的部分原因。人总是喜欢活跃在自己的舒服区，这也是人之常情。但要想做好架构师，就要适当地走出舒服区。

架构师，也就是软件开发组织的领导，并非有架构师头衔的那个人。架构师应该承担起解决业务问题的角色，专注于业务和软件本身的架构，通过对软件开发生命周期和软件运行生命周期的拆分，让技术人员合理地分工。只有把业务和软件很好地结合起来，才能更好地完成业务目标，才会让软件更好地服务于大家。这样执行下去最终一定会得到一个很好的软件架构，使得软件开发团队和业务部门都能够很好地开展工作，达到软件和业务的增长目标。

16.4　重新发明轮子

当现有已经存在很多技术，而这些技术所解决的问题，和业务所要解决的问题，并没有直接对应的时候，就需要有意识地拆分业务的生命周期。当拆分出来的非核心生命周期足够小时，慢慢就会出现符合要求的技术，直接采用即可。可是当拆分还不是那么确定，但是业务的问题又亟待解决，那么应该采用哪些技术，

还是自己重新实现一个呢？ 重新实现一个，就是很多人所谓的重新发明轮子。下面就对几种不同的情况分别讨论一下：

（1）当技术所要解决的问题和拆分出来要解决的问题完全匹配时，这是最完美的。比如需要提供 Web 访问的服务（Service），而很多 MVC 的框架（Framework）就是解决这个问题的，可以很好地满足这一点。这种情况如果非要重新实现一个，很有可能是重新发明轮子。

（2）当技术所提供的能力远远超过需要解决的问题时，往往掌握技术和维护技术就会成为负担。越复杂的技术，成本越高，带来的问题也会越多。如果自己实现一个仅仅是解决当前问题的技术，可能成本反而更低。

（3）当技术所提供的能力和我们所要解决的问题部分匹配时，要判断是否要采用，最终还是要看成本。比如当人们需要一个锤子的时候，手边正好没有，但是却有一只高跟鞋，勉强也可以用来替代锤子来解决问题。但是长期这么用不划算，因为高跟鞋的价格比锤子高很多，耐用性却差很多，而维护成本也高很多。

所以，在架构拆分的基础之上，识别并平衡技术的能力，也是架构师所要具备的能力之一。考虑的主要因素是长期的成本和收益。

16.5 开源技术

软件行业有太多人们没有遇到过的问题，比如用户的访问流量疯狂上涨就是其中之一。这也是为什么很多大型的互联网公司不断地开发出新技术的原因：他们所遇到的问题并没有现成的技术框架能够解决，其他公司也不一定遇到过。这些公司会把他们所创造的新技术开源出来，贡献给社区。

很多人无法理解，为什么很多企业轻易地就把代码公开出来？这不是把自己的核心竞争力暴露出来了吗？

要知道开源的只是代码而已，而代码并非是软件生命周期的核心。软件生命周期的核心是代码的运行生命周期和用户访问生命周期，而不是代码的建立生命周期。很少有企业会把自己的软件运营体系拿出来说事，企业对运维区域的管理特点往往都是以严防死守为主的，类似于军事化管理。因为软件运行生命周期这部分才是软件真正的核心能力。

代码开源就好比是很多人通过写书来阐述自己的理念，因而代码开源和出版一本书的含义类同。读者看了书并不意味着就能把书中的理念变成自己的，马上运用自如。开源的代码也是如此，软件工程师拿到了别人开源的代码，就相当于拿到了别人写的书，并不代表软件工程师就能够理解别人的代码。即便理解了别人的代码，也不代表能把软件运用好。

对于开源者来说，开源是把自己的理念介绍给外部的一个手段，和写一本书是一样的。区别之处在于，写一本书是在读者大脑里运行的，而代码是在计算机中运行的。相同之处在于，要想让代码在计算机里运行好，读者仍然需要理解作者遇到困难时，为何这样思考，找到作者解决的问题独特思路。也就是说，代码最终还是要和书一样在读者的大脑里运行起来，才能够把代码所产生的软件运用好。

明白了以上道理就不难理解，代码的核心在于作者对其理念的实现，而不是代码本身。代码的内容反映的是作者对世界的认识，反映的是作者的世界观。就好比拿到一本修炼绝世神功的秘籍一样，并不是所有人都能够修炼成功。即便读者看懂了代码，也不代表神功就能在自己的身体内发挥作用。要想让神功发挥作用，必须要先把自己的观念转变成和作者一致才可以。

修炼的过程就是把自己的思维观念变成和作者的一致的过程。

代码编译好了可以运行，相当于拿到了绝世武器，但这不代表你能够称霸天下、能够解决问题，有时搞不好还会弄伤自己。

因此要想采用这些开源技术，就要理解这些开源技术往往都有其特定的使用场景。比如针对大流量的技术，对于其他小公司来说，大多时候是过于复杂的，采用之前必须了解这些技术原本是用来解决谁的问题的？什么问题，如果不先搞清楚就冒然采用，很有可能会导致更高的研发成本。小流量的公司采用针对大流量的新技术，仅仅维护和监控就足够让人头疼。若环境配套跟不上，则技术是无法发挥作用的。而和新技术最重要的配套就是人的配套，以及人的观念的配套。

作为开源技术的使用者，要掌握好作者的理念，一般要先从理解作者面对的问题入手，也就是从业务入手，分析作者是如何拆分业务生命周期的。从这个角度来说，看代码和看书没有区别，只不过一个是用计算机语言写的，一个是用人类语言写的，展现的形式不同而已。

所以根本不用担心开源代码会暴露自己的核心竞争力，开源只会带来更多的好处。一方面全世界的软件工程师都可以帮助改进所开源的代码，相当于收获了一个巨大的开发团队；另一方面也是对社区进行贡献，毕竟：

要先付出才能有收获，这是这个世界运行的规则。

当开源成为一个趋势时，所有人都可以从中获益，包括开源的企业自身。人们可以从社区的开源技术中获取营养，或者使用别人开源的技术。同时，全世界的软件工程师形成了一个新的分工，分工不再局限于软件工程师各自所在的企业，而是变成了一个全世界的软件工程师组织架构，共同推进、解决软件行业所遇到的各种挑战和问题。

第 17 章　软件研发

软件的生命周期可以拆分为软件开发生命周期和软件运行生命周期。软件的开发生命周期结束后，生成的是软件，紧接着软件运行生命周期就开始了。一个完整的软件开发生命周期推进的过程，一般称之为软件的研发。执行软件开发生命周期的人员，称之为软件研发人员。参与一个软件的所有研发的人员的集合，一般称之为软件研发团队。一个软件研发团队的领导人，就是这个软件的架构师。软件研发的整个过程也是软件本身要虚拟化的业务领域之一，因此有必要讨论一下。

17.1　软件工程师的兴起和使命

软件最早更多的是应用在科学计算方面，有些大规模的计算，如果用人来算，则需要花很长的时间。研究人员为了提高自己的工作效率，把自己的工作写成软件来自动化。计算机计算的速度很快，并且不用休息。也就是说，软件最初的编写人员本身就是业务专家，业务非常精通，然后才学习用软件来提升业务的。至今，科学领域的研究人员大都还是自行编写自己领域的软件。

随着计算机成本的下降，计算机开始逐渐普及，很多行业都发现可以用计算机和软件来提升自己业务的效率。但是这些行业的从业人员大多都不了解计算机，更别说软件了。怎么办呢？

一个办法是，让这些行业的从业人员自行去学计算机和软件，与科学领域的研究人员一样，自行掌握计算机和软件的的技能，改造自己的行业。但是计算机和软件的学习门槛对于大部分行业而言是比较高的，是一门跨学科的专业，建立在很多的学科之上，如数学、物理、电子电路等。科学领域的研究人员为什么可以做到呢？因为研究人员有这些学科的基础，学计算机和软件相对会比较容易，对他们而言门槛很低。

另外一个办法是，吸引能够编写软件的人为本行业服务。这是可行的，大部分行业，锻炼一两年的时间就可以比较熟练了。因为有这个巨大的需求在，慢慢地：

编写软件就从科学计算中拆分出来，变成了一个通用的服务，成为了一个独立的职业，具有独立的生命周期。

这就是软件这个新技术的出现，导致了新的社会分工和新的架构拆分。

从此软件不再专门服务于科学领域，而是服务于各行各业了。这个职业的从业者，也就是软件工程师，大部分只有计算机和软件的专业知识，并不懂计算机之外的业务。他们要去解决行业的问题，就必须要学习并掌握行业的知识。所以软件工程师的知识结构和科学领域的研究人员不一样：软件工程师先学习计算机和软件知识，再学习行业知识，并且软件工程师对行业知识的掌握往往不像研究人员对科学领域所掌握的那样精通。

很多软件工程师学了大量的算法和计算机基础，然而在工作中发现派不上用场。这是非常正常的，因为这些内容是为了在科学领域做研究准备的。而在业务领域，大多是如何把现有的业务在软件中模拟出来的问题，并没有太多高深的数学问题。并且现在的计算机硬件，比如 CPU、内存、存储等都非常便宜，也不需要斤斤计较地去抠时间和空间复杂度。这些都导致所学不能致用。反而如何能够高效地把业务用软件表达出来，并能够随着业务的增长，让软件也快速地长大，则变成了一个更重要的问题。这一点可能是当前计算机软件教育需要思考的问题。

对于传统企业而言，业务长大主要是靠增加人与空间，成本很高。但这是传统企业所熟悉的方式，心里有底的方式。在软件工程师慢慢理解并掌握了业务后，慢慢就会看到传统企业各种低效的地方，也会尝试着去改变现状。而传统企业的思维是很难转变的，业务方通常只是把软件作为一个帮助自己的新技术、新工具而已，并不把软件看成一个新型的虚拟人，甚至以软件来代替业务人来工作。

如果每做一件事，都需要业务点头，这对于受过高等教育的软件从业人员而言是很难接受的。软件工程师和业务人员低效率的沟通，以及长期的冲突都证实了这一点，导致软件的产出质量都不是很高。传统企业的从业人员无法掌握软件，而一旦软件工程师掌握了业务，慢慢软件工程师就会脱颖而出，开始用软件的虚拟化思维思考业务。这部分人慢慢就会开始脱离传统企业的发展轨迹，直接在软

件中模拟业务，在虚拟世界进行更高效的扩张，进而形成了对传统企业的颠覆。互联网电商，互联网金融等都是软件虚拟化的现实例子。

软件和业务最终还是要合体的。

当前软件工程师专职开发服务于业务的形式只是一个过程。经过这个过程的普及，掌握业务的软件工程师未来会越来越受欢迎，各行各业掌握软件开发的业务人员也会越来越多，软件也会慢慢变成平常老百姓的必备技能，就像现在的智能手机一样，人人必备。在这个过程中，传统企业需要主动地去接纳会业务的软件工程师，并且培养软件工程师做业务，降低沟通的成本，赢得未来。同时，软件工程师也应该跳出工程师的视角，以业务的视角来思考，则可以更快地融入业务，并主导业务。

虽然前面说传统行业的软件虚拟化是对传统行业的颠覆，但是业务本身的规律是不变的。比如前面所说的互联网电商、互联网金融等，其行业基本规律并没有改变，区别在于用户访问企业的生命周期发生了新的拆分，从现实空间转到了虚拟空间；其长大的方式从以人和空间为主，变成了以计算机和软件为主；而计算机和软件长大的成本要远远低于人的增长，虚拟空间的增长成本要远远低于真实空间的成本。成本一旦降得非常低，原来很难甚至不可想象的事情就变成了可能，甚至很容易了。因此可以极大地提升整个社会的效率，让人们的生活变得更加便利，有更多的时间来做提升自身生命价值的事情。

17.2　分工的困境

软件工程师进入某个行业是有一个过程的。对于某个业务组织而言，以传统企业为例，往往分工已经较为成熟，亟需软件来进一步提升生产力。刚开始一般都是先试水，引入几个工程师，开始把一小部分的业务软件化、自动化。这些工程师进入企业后，一般都是先去和业务人员学习业务知识。熟悉业务后，再根据业务人员面对的困难，把业务用软件代码表达出来，形成软件。在业务人员对软件表达出来的业务确认后，上线运行，完成软件的创建。

而对于企业来说，由于分工太细，企业的员工们往往不满足于扮演“螺丝钉”角色，做这些重复枯燥的工作。卓别林的经典电影《摩登时代》可以看到分工细化后对人的影响，现代的工厂化生产仍然存在同样的问题。比如一些现代工厂招

工时，甚至要做智商测试，用来排除智商较高的候选人。理由是智商太高，会很快对重复枯燥的工作厌烦而辞职。因此，把企业的业务自动化，把员工从粗糙、重复的工作中解放出来，释放更大的生产力，成为一件必须要做的事。而软件则在业务自动化中扮演了至关重要的角色。

以典型的互联网企业为例，往往连业务都还没有开始分工，一个软件就包办了几乎所有的事情。几个软件工程师包办了几乎所有的工作，包括业务运营、软件编写。也就是说，在这个时候，软件工程师和业务是合体的。软件工程师的活动基本上就是思考业务的场景，把业务场景用软件表达出来，上线运行，完成软件的创建。然后再通过用户的反馈，不断地修改软件。一旦业务人员和软件工程师合体之后，效率往往会上升几个层次。

所以，无论是传统企业，还是现代的互联网企业，软件的开发生命周期基本上都包含以下几个关键的活动：

- 确定业务需求
- 编码
- 测试
- 上线

这些活动按顺序发生，每一个活动都是下一个活动的前提，而且必须要严格按照时间顺序推进，才能最终得到结果。这是一个典型的生命周期过程，通过不同的活动，随着时间的积累，直到软件上线作为结束。当业务发生了变化之后，还要对软件进行修改，这个软件会再次经历一个完整的开发生命周期。和上一个开发生命周期的区别在于，本次生命周期的起点是上一次开发生命周期的果实。也就是说，一个个开发生命周期本身也是按照时间顺序发生的，这些开发生命周期组成了软件的生命周期。

在软件开发的早期，没有互联网，只有传统企业。软件刚形成独立的生命周期，大部分从业人员对软件的生命周期都不是很了解，此时大家都采用科研人员的方式来推进软件生命周期。整个软件开发生命周期基本只有一个人来推动执行，但一个人的能力是有限的。如果要在某个限定的时间内做一个大型的软件，则必须要增加软件研发的产出，势必要引入更多的人。而引入很多人做同一个软件，每个人又都去写代码，既不现实，也不具备可操作性，因为没有办法进行分工。

此时没有服务化，也没有软件的拆分。要分工，自然而然就会分不同的角色做不同阶段的事情：有些人做需求、有些人写代码、有些人做测试，等等。这样分工容易，招人也容易，分出来的工作门槛也会比较低。

但当引入不同角色来分别做不同阶段的事情时，就会出现一个问题：在某个角色所负责的前一个阶段工作没有结束之前，负责下一个阶段工作的角色没办法开始工作，因为下一个阶段要等待前一个阶段工作的结果才能开始。也就是说，在整个生命周期活动中，同一个时刻只有一个环节是活动的，实际产出是 1+1=1。并且沟通过程中还会造成信息的损失。比如前一个阶段工作的负责角色，要把其所负责阶段工作的成果传递到下一个阶段工作的负责角色，会造成沟通上的偏差。最后得到的结果其实是 1+1<1，增加人之后的效率反而更低了。这就是传统的瀑布式软件开发模式。

瀑布式开发模式本身是没有问题的。比如在建筑行业，其实就是瀑布式的模式，在需求做完后进行设计，在设计完成之后，再开工建设。区别在于，建筑行业是对空间切分，而空间是现成的。只需要顺从地球的自然地理环境以及地球的重力等，大部分都是可预测、稳定的。建筑设计师本身也是建筑的使用者，对用户的需求感同身受，容易理解。而且只有在开工后才需要大量的工人，这些工人的门槛很低，很容易就能够招到，招不到也比较容易就能培训出来。因此在建筑行业这么运作并没有什么问题。

而软件行业则不一样，软件是用来代替业务人员的，首先要先理解业务人员的工作，才能够做出设计。而理解业务人员的工作会存在各种各样的困难，特别是沟通的偏差会导致软件的起点就会有问题。更别说软件是用来替代业务人员的，业务人员也会有意无意地抵抗这一点。软件工程师本身的门槛很高，因此想招到合适的人也非常困难，培养一个人也很困难。所以软件行业采用瀑布式模型时，就会存在大量的浪费，最终的结果也往往不尽如人意。

17.3　软件的迭代

软件开发生命周期本身就要求前一个阶段完成才能够产生下一个阶段，因而瀑布式的时间推进过程是无法避免的，也不应该避免。因为毕竟要通过生命周期活动的不断积累，使得软件不断的长大，成长为可以为人类带来收益的虚拟人。

因此有必要针对瀑布式开发模型出现的问题作相应的分析。

首先是软件开发工程师和业务分离的问题。瀑布式的开发模型通过强调文档的约束来解决两者的沟通问题，这是远远不够的。真正的问题是文字无法描述的，必须亲身体会才行。所以慢慢的有软件架构师从软件工程师中成长出来，亲身体会业务、掌握业务，对业务的生命周期进行分析，并应用到软件中去。这一分工和建筑就很相似了。

软件也不像建筑那样可以用图画出来，或者以一个模型展示出来，给人以直观的体验。软件看不见、摸不着，必须运行起来，通过计算机的输出设备与之交互，用户才能知道软件是否达到了预期的效果。

这就意味着，软件无法一次性的把所有的业务都模拟出来，必须要分步骤的一个一个阶段做出来，通过用户的反馈，一点点的长大。这就是所谓的迭代。每一个小的迭代都是瀑布式的推进，每一个迭代到下一个迭代也是瀑布式的推进，整体仍然是瀑布式的推进。所以瀑布式的推进，也就是活动按时间顺序发生，本身是没有问题的。即便是建筑，也是一层一层从最底下开始迭代造出来的。

迭代应该如何进行呢？迭代的前提是，必须要先确定优先级，而理清楚业务的核心生命周期是最高优先级。就如前面介绍的“钻木取火”的例子，首先要实现业务的核心生命周期，刚开始可以简陋一点，效率可以低一点，此时一般也不需要太多的人参与，成本也不会太高。一旦业务的核心生命周期模拟出来之后，业务人员才可以操作软件，对软件进行反馈。形成反馈环后，软件就可以通过一次一次的迭代，丰富核心生命周期的功能；或进行架构拆分，形成新的非核心生命周期；或通过反馈，发现之前的核心生命周期识别错误，再进行重新调整。软件就逐渐长大了。

软件开发的生命周期要允许犯错，因为软件模拟的是人，人与人合作总是会因为沟通等问题犯错的。人与人之间必须通过亲身体验并操作才能够发现错误，仅仅依靠书面和口头的沟通是无法避免错误的。要减少错误，就必须要犯错误，只要犯的错误能够得到快速地反馈和纠正就不会造成问题。相反，犯的错误越多，纠正得越快，就越能够减少线上的问题。要想快速发现和纠正错误，就要做好迭代，而做好迭代就必须确保迭代的生命周期要足够短，这样就可以快速地纠正错误，才能够拥抱错误。这就是所谓的“敏捷”（Agule），核心就在于快速迭代并得到反馈。

这也引出来了瀑布式开发模型的另一个问题，也就是一个软件开发周期要实现的内容大多，导致周期太长。其周期往往以年为单位，由于无法及时地反馈并纠正问题，因此会尽可能地排斥错误，而排斥错误并不一定能让错误减少。快速迭代则以月，甚至周为单位，这样一旦发现问题，或者业务有变化，就可以快速地应变，甚至“试错”。

所以生命周期活动按时间顺序发生的特点，本身并不是问题，而控制好一个周期的长度才是关键。

17.4　软件开发的分工

软件开发生命周期的核心人物是软件工程师。因为软件开发生命周期的产出物是软件，而软件是由代码组成的，代码是由软件工程师编写的。所有的软件开发生命周期的活动，以及软件开发过程中的所有其他角色，都是为了帮助软件工程师更好、更有效率地编写代码，或者修改代码。

为了组织好代码，架构师需要去理解业务的核心生命周期以及业务的架构拆分，形成代码的架构。

为了组织好软件工程师，我们需要把软件划分为组件，或者拆分软件，尽可能的让软件工程师之间并行工作，减少冲突。

为了组织好软件工程师，就需要形成软件工程师的组织架构，与软件、组件进行对应，与业务的组织架构对应。

因此，架构师成为了软件开发生命周期推进的组织角色，为软件工程师们保驾护航。从这里也可以看出：

软件架构师并不一定是具备架构师头衔的那位，但一定是软件开发团队组织的领导。

综上所述，架构师要做的事情非常多，因此也需要很多不同角色的帮助，拆分自己工作的生命周期。

比如要进行迭代，一个迭代会根据业务需求组织相应的资源。这些资源以人为主形成组织架构，这个组织一般被称为项目。对于一个软件而言，就会有多个迭代，也就是多个项目并行进行。因此架构师就需要项目经理来协助搜集每个项

目，也就是每个迭代的进行状况，对不同项目做不同的调整。

一旦软件由多个工程师开发，那么软件工程师自己电脑上的代码就不准确了。此时需要一个代码管理服务端，所有软件工程师的代码都需要提交到这个代码管理服务端上，以这个服务端的代码为准。因为软件有多个项目并行，自然就导致代码会有多个分支，多个版本，这就是软件的代码管理。软件的代码管理就从软件开发生命周期中拆分出来，形成一个新的生命周期，由独立的团队进行管理，以服务的方式提供给软件工程师。

当软件的代码管理独立出来后，软件的构建就不能以软件工程师的电脑为准了。因此，软件的构建也会从软件开发生命周期中拆分出来，以软件代码管理服务端为代码源，形成新的软件构建生命周期，由独立的团队进行管理，以服务的方式提供给软件工程师。

同样，软件的部署生命周期也会独立出来，以软件构建生命周期的输出为源，由独立的团队进行管理，以服务的方式提供给软件工程师。

还有，软件的测试生命周期也会独立出来，以软件构建的部署生命周期的输出为源，由独立的团队进行管理，以服务的方式提供给软件工程师。

等等等等，还有其他更细的拆分，就不一一展开了。

从这里可以看出，软件开发生命周期本身就是一个比较复杂的业务，自身也需要软件来进行虚拟化。因此也可以看到大量的工具，如软件需求管理、项目管理、代码管理、测试管理、构建管理、部署管理，等等，把软件开发的核心生命周期精简，把非核心生命周期拆分，用软件来实现自动化，让软件工程师能专注于写代码，提升软件工程师的效率。这个过程和软件帮助业务是一样的。因为：

软件开发业务本身也是软件的一个业务。

17.5 软件开发模式和架构

从上文的分析中可以看到，软件研发是一个复杂的活动，是创造并组织虚拟人的活动。为了解决这个复杂的问题，或者说这个复杂的软件业务，人们发明了很多不同的方法论和开发模型，并做了很多的实践，寻找所谓的“银弹”。无论哪种开发模型，基本上都可以分为以下两种类型：

（1）以不信任软件开发工程师为基础，以避免软件开发工程师犯错为核心的开发模型。

（2）以信任软件开发工程师为基础，以软件开发工程师为核心的开发模型。

早期的开发模式基本都是第一种，也是传统企业经常采用的方式。由于不信任软件工程师，因而所有的需求和沟通都要求采用文档来传递；不要求软件工程师理解业务，而是要求软件工程师严格按照文档实现；并且建立庞大的测试团队，防止软件工程师犯错。因此开发流程往往非常复杂，开不完的会议、写不完的文档，而结果经常达不到预期。进而导致更多的流程、更多的会议和更繁杂的文档，恶性循环，参与者苦不堪言。

为什么会这样呢？这就是因为业务人员和软件开发工程师分体的缘故。业务人员是出资方，有全面的业务话语权，可是软件开发的核心是软件工程师，对软件有绝对的话语权。两个话语权分散了，权责不对等，导致软件工程师没有办法真正地理解业务，开发出的软件自然就长得歪歪扭扭变形了。而让业务方完全放手是不太可能的，这就是为什么传统企业转型这么难，要给自己“捅一刀”才行。

新兴的互联网企业则往往会选择第二种方式，信任软件工程师。一旦把业务建立在软件之上，软件工程师的作用就至关重要。毕竟软件工程师是唯一真正通过代码构建软件的角色，必须要绝对的信任。这个方式形成了一种新型的企业文化，也就是“工程师”文化。这是正确对待软件工程师的方式，而这个方式必然就要求软件工程师掌握业务。业务和软件两者一合体，就产生了巨大的生产力。

如果软件工程师对业务一知半解的话，则软件是不可能写好的。软件开发的整个过程，参与其中的的所有角色，都应该以软件工程师为核心，帮助软件工程师理解业务，让软件工程师成为业务专家。这听起来似乎又回到了科学研究领域的研究人员采用软件的方式。确实是的，只不过研究人员是先学业务再学软件，而软件工程师是先学软件再学业务，顺序不同而已。只有软件工程师成为业务专家，写出来的软件才是靠谱的。

如果业务简单，流量足够小，那么几个软件工程师就可以了，并不需要其他角色参与。比如一个创业公司，可能几个软件工程师就搞定了。而一旦业务复杂度比较高，或流量上了规模，就需要很好地组织软件工程师的分工，此时就需要架构师来帮忙了。

17.6 软件工程师的支持者

在所有帮助软件工程师进入业务的角色中，最重要的角色是软件架构师，也就是软件工程师团队组织上的领导。软件架构师除了管理软件工程师外，其他帮助软件工程师的角色也要统一给软件架构师管理，确保这些角色能够配合软件工程师，以软件工程师为核心开展工作。软件架构师要学习业务的架构，根据业务特点确定软件开发的核心生命周期，根据业务组织架构确定软件的拆分和架构：

组织软件工程师进行有效的分工，形成软件开发团队的组织架构。

为了减轻软件工程师的负担，要把软件开发过程中的非核心生命周期拆分出来，形成软件开发本身的业务架构，并通过个自动化来提升软件工程师的开发效率。

这些架构都是树状的架构，都是对生命周期拆分所形成的。所以软件架构师要想做好架构工作，首先必须是业务的架构专家，同时还得是软件的架构专家，这样才能够支持好软件工程师的工作。

架构师如果自身时间有限，则必须要对自己的工作进行架构切分。根据自身的核心生命周期，把非核心生命周期拆分出去，交由其他的角色负责，所形成的也是一个树状架构。也就是说：

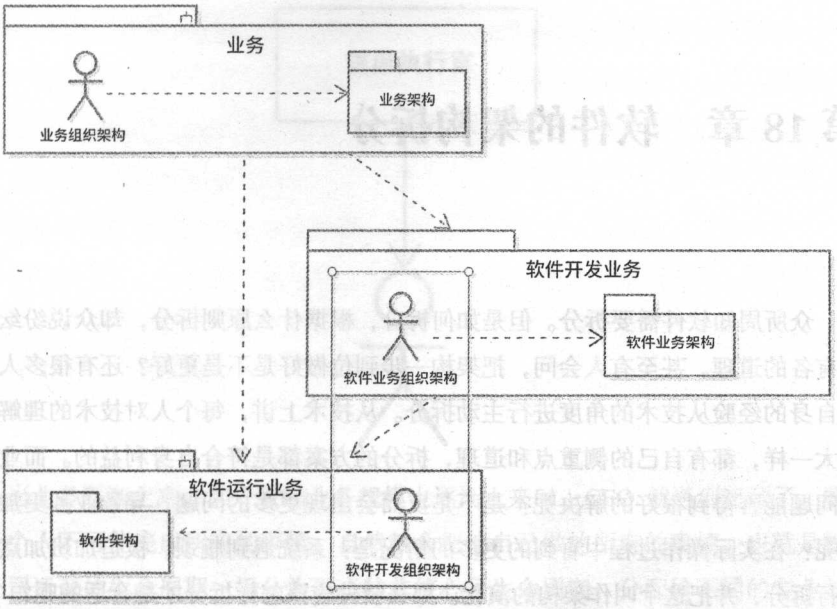
架构师本身也是架构师的一个业务，也需要架构拆分，形成不同领域的架构师。

综上所述，软件所面对的共有三大业务领域及其所对应的架构，如下图所示：

（1）业务领域，由业务组织架构来推动业务的架构，即业务生命周期的拆分；

（2）软件开发业务领域，由软件开发的业务组织架构来推动软件的业务架构，如软件的研发流程、角色分工等。所形成的是软件开发模式，不同角色的分工模型；

（3）软件运行业务领域，由软件的开发工程师来负责编写代码，形成软件的架构，并支撑软件的运行。对不同的软件开发工程师进行分工，形成不同的软件开发工程师组织架构，以支撑不同的软件，与软件的架构相匹配。



软件业务组织架构和软件开发组织架构共同组成了软件研发团队。业务的特性和软件开发业务的特性同时累积到软件中，形成了软件的架构。软件架构师们需要注意这个影响的链条，这是软件的整个大环境，三者的变化都会随时随地影响着软件的架构。

由此可见，在软件开发生命周期中，软件工程师和软件架构师是最重要的两个角色。他们的分工分别是：

软件工程师负责建设，软件架构师负责组织。

为了支持软件工程师的工作，软件架构师的主要职责包括以下几点：

（1）理解业务组织架构，业务组织架构支撑并推进业务架构，背后的原因是对业务生命周期的拆分。

（2）根据业务生命周期的特点和软件开发生命周期的特点，形成了软件开发本身的业务体系，以及对软件开发生命周期的拆分，也就是软件开发的业务架构。

（3）根据对业务生命周期的以及软件开发生命周期的拆分，形成了和两者都相匹配的软件开发团队的组织架构。

（4）对软件进行架构拆分，匹配业务架构和软件开发的业务架构。

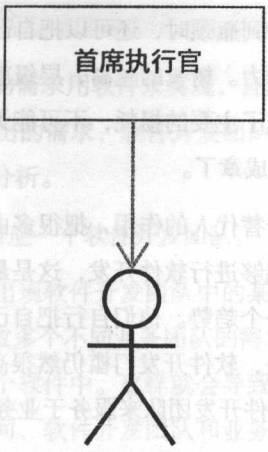
第 18 章　软件的架构拆分

众所周知软件需要拆分。但是如何拆分，根据什么原则拆分，却众说纷纭，各有各的道理。甚至有人会问，把架构一步到位做好是不是更好？还有很多人根据自身的经验从技术的角度进行主动拆分。从技术上讲，每个人对技术的理解都不太一样，都有自己的侧重点和道理，拆分的方案都是符合自身利益的。而业务的问题能否得到很好的解决呢？是不是反而会出现更多的问题，导致业务更加恶化呢？在实际操作过程中看到的更多的情况是：系统遇到瓶颈，被迫加班加点地进行拆分，并把这个叫作架构的演化。那么软件应该怎样拆分才是合适的呢？

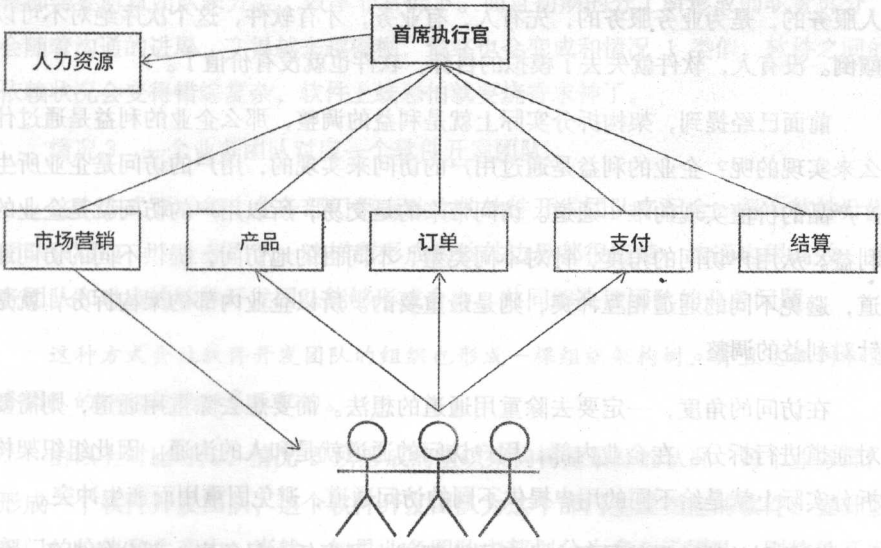
18.1　软件拆分的原动力

软件架构拆分的原动力实际上是来源于业务本身的拆分所形成的组织架构。这个似乎比较难理解，不妨先从业务本身开始分析。业务的组织架构是怎么形成的呢？业务组织架构拆分的背后原动力，是每个人的负载超限。这个负载主要是人的时间不够，无法在一个人有限的时间内完成大量的业务，就只能把整个工作这个大的生命周期，切分为小的子生命周期，形成一个树状架构。然后把切分出的不同非核心子生命周期授权给具备不同专业技能的人，让他们对各自所负责的生命周期负全责。而拆分的人则成为了整个组织的领导，负责把各个子生命周期的运行结果组合为自己的产出，这样他一个人在相同时间内的产出就放大了许多倍。

比如一个企业刚开始时，CEO（首席执行官）一个人处理所有业务，直接和用户打交道。这个时候客户比较少，问题还不大。



当业务逐渐上来，客户量和业务规模也逐渐上来时，CEO 就遇到瓶颈了。既然一个人不可能完成所有的事情，自然就会把他自己做的所有的事情，也就是整个公司运营的生命周期，拆分为更小的非核心子生命周期，分配给不同的专业的人员来运营，而自己只把控核心生命周期。其他人则负责推动非核心子生命周期，围绕核心生命周期负责人形成一个树状的架构，共同为企业的生命周期服务。如下图所示。



当每个专业领域自身遇到瓶颈时，还可以把自己的生命周期细分成更小的生命周期，以提升自己的生产力。树长的越高，层级就会越多，但是当增长到一定程度的时候，沟通成本就成了主要的损耗，不可能无限地增长下去。所以引入软件技术来提升生产力就顺理成章了。

软件实际上是扮演一个替代人的作用，把很多由人执行的事情交给软件来模拟并执行。如果业务人员能够进行软件开发，这是最好的情况，就如科研领域的人一样，这在未来应该是一个趋势。他们自行把自己的职责自动化，可以提升自己的生产力。但实际情况是，软件开发门槛仍然很高，普通的企业还做不到这一点，所以必须成立独立的软件开发团队来服务于业务人员。

以上看起来好像说的是传统企业采用软件的过程，但实际上是想说明企业背后架构发展的逻辑，并非特指传统企业。互联网企业的发展也是如此，其和传统企业的区别在于，互联网企业一开始就专注于采用软件来提升生产力，业务团队的增长往往会慢于软件体系的增长。甚至更彻底一点的企业，软件工程师和软件架构师本身就是业务人员。由于软件强大，业务人员的生产力提高得非常快，不需要大量的业务人员投入，但需要大量的软件开发投入。但是绝对不可以因为没有经历无软件的纯业务期，而忽视业务团队，放弃以业务为中心。因为软件是为人服务的，是为业务服务的，先有人，有业务，才有软件，这个次序绝对不可以颠倒。没有人，软件就失去了模拟的目标，软件也就没有价值了。

前面已经提到，架构拆分实际上就是利益的调整，那么企业的利益是通过什么来实现的呢？企业的利益是通过用户的访问来实现的，用户的访问是企业所生产产品的价值实现的唯一通道。访问带来的是交易，所以用户的访问就是企业的利益。从用户访问的角度，针对不同类型、不同目的地访问，提供不同的访问通道，避免不同的通道相互冲突，则是最重要的。所以企业内部的架构拆分，就是针对利益的调整。

在访问的角度，一定要去除重用通道的想法。而要想去除重用通道，则需要对通道进行拆分。在企业内部，用户访问的通道就是和人的沟通。因此组织架构拆分实际上就是给不同的用户提供不同的访问通道，避免因重用而产生冲突。

调整利益的结果就是每个业务团队都减少了责任，只负责一部分的业务，每个团队的成长空间提升了。对外访问通道的拆分结果是形成一个树状架构。

18.2　软件开发团队的拆分

因为要把业务部门提出的需求用软件来实现，自然就需要建立软件开发团队。那么对于不同的业务团队提出的需求，软件开发团队又应该如何分工组织来应对呢？对此分几种情况来进行分析。

情况 1，多个业务团队对应一个软件开发团队。

在这种情况下，很容易出现软件开发团队中的某些人会处理多个不同业务团队的问题，进而就很容易导致多个不同业务团队的需求，以省力或者重用为理由，全部或者部分地整合在同一个软件中。这样就会导致这些业务部门都在提需求的时候，软件开发团队内部之间，软件开发团队和业务部门之间，会产生大量的互相依赖、干扰、扯皮，也会产生大量的会议，降低整个团队的效率。最终这些软件之间的依赖状况会变得错综复杂，软件上线则要全看运气了。

情况 2，一个业务团队对应多个软件开发团队。

在这种情况下，业务团队必须要自行对业务需求进行拆分，以便给到相应的开发团队。而业务团队一般都不具备这个能力，只能同时和多个软件开发团队一起开会讨论。这同样会产生很多的沟通、扯皮。并且这种沟通是持续的，每次调整都需要所有团队来开会，效率非常低下。而且初期的分工所形成的职责拆分，会随着沟通的进展，变得越来越模糊，最终也会变成和情况 1 类似。软件之间的依赖状况会变得错综复杂，软件上线恐怕就要烧香求神了。

情况 3，一个业务团队对应一个软件开发团队。

这个方式要求每个业务部门都有独立的软件开发团队来配合。每个软件开发团队只对应一个业务团队，这样所形成的软件边界都很清楚，沟通也很高效。业务团队和对应的软件开发团队能够形成合力，共同解决该团队的业务问题。

这种方式会让软件开发团队的组织也形成一棵组织架构树，并且这棵树和业务团队的组织架构树是匹配的。

由以上对比可知，情况 3 所形成的组织架构树是最好的状况。每个业务部门形成一个软件开发团队，这个软件开发团队为这个部门生成相应的软件，提升该部门的价值和生产力。当然，如果业务团队内部划分为多个子团队，则软件开发团队也应该一样划分相应的内部子团队，一一对接，生成相应的软件。从这个意

义上来说，软件开发团队其实应该是和业务部门在同一个部门的，都是为同样的业务服务。当业务组织发生变化的时候，软件开发部门也应该要有相应的变化，否则就会变成情况 1 或者情况 2 的结果了。这就是业务组织架构对开发团队组织架构的影响。顺从这个影响，沟通协作就会很顺。反之，则需要大量的资源来纠正这些问题，表现出来的就是冲突频繁，内耗严重，每个人都很累，但是成效却不大。

同样，软件开发团队的拆分也是利益的调整。软件开发团队的利益来源于哪里呢？软件开发团队的利益来自于对业务部门需求的实现，也就是来源于业务部门对软件开发团队的访问。业务部门就是软件开发团队的用户，不同的业务部门对软件开发团队的需求是不一样的，访问的目的和要求也都不一样。因此，业务团队对软件开发团队的访问通道也不能够重用，避免不同业务部门的访问互相冲突。

软件开发团队对业务的访问通道是人，为了提供不同的访问通道，就必须要对软件开发团队进行拆分，让不同的人提供不同的访问通道，避免重用产生的冲突。上面的三种情况已经解释了这一点。

调整利益的结果就是每个软件研发团队减少了责任，只负责一部分业务。但每个团队的成长空间更大了，沟通的效率也提升了，也更专注了。对外访问通道的拆分结果，形成的还是一个树状架构。

18.3 软件的拆分

同样，软件是软件团队的产出物。所谓的软件，是指一个可以单独部署运行的完整产出物。对于软件开发团队和软件的关系，有以下三种情况：

情况 1，多个软件开发团队开发同一个软件。

这就是属于共享访问通道的情况。这时对代码的分支合并的管理要求非常高，经常会发生互相把对方的代码冲掉的情况，导致严重的线上事故。当某个软件需求较多，或者某个团队的版本出问题的时候，就会导致排队上线的情况，拖慢对业务的响应。

情况 2，一个软件开发团队开发多个软件。软件开发团队内部会很难形成明确

的分工，导致一个人会同时修改多个软件。这样就会导致同一个需求，在既可以放在 A 实现，也可以放在 B 实现的情况下，让多个软件的边界变得非常模糊，形成不恰当的依赖。

情况 3，一个软件开发团队开发一个软件。这样每个软件的职责非常明确，沟通也会比较简单，这是最好的状况。

这时形成的还是一棵树。软件和软件之间的关系，反映的就是组织和组织之间的关系，一一对应，还是一棵树。

当软件开发团队发生拆分时，软件也需要进行相应的拆分，否则就会变成情况 1。当开发团队发生合并时，软件不一定合并，这个时候会比较难处理，很容易就会变成情况 2。所以一般软件团队开始合并时，往往都是比较混乱的。这个时候一定要确保内部的原有分工得以保持，除非软件也合并。

在软件开发团队内部，精确到每个人的分工也是一样的道理。软件的内部代码也会进行架构拆分，拆分出来的每个部分都被称为组件。组件一般是一个可独立开发的单元，不可以单独运行，用来做软件开发人员分工的单元。软件可以由一个或多个组件组成，形成一棵依赖树。一个组件根据内部的分工，可以分成不同的职责：有的负责业务的开发，有的负责技术的开发，这同样会导致组件的拆分。这里限于篇幅，就不一一展开了。所以：

从软件团队到人，从人到组件，形成的还是一棵树。软件和组件，组件和组件之间，形成的也是一棵树。

同样，软件的拆分也是利益的调整。什么，软件也有利益吗？是的，前面已经解释过，软件就是一个虚拟的人，所代表的就是业务人员，因此也是有利益的。

软件的利益通过什么来实现呢？软件的利益通过用户的访问来实现，和业务人员的利益保持一致，因为软件模拟的就是业务人员。对业务人员的拆分，自然就是对软件本身的拆分，避免不同的用户访问通道相互影响。因此，软件在访问通道问题上也不能重用。

重用访问通道的结果，既损伤用户的利益，也损伤软件的利益，还会损伤软件开发团队和企业的利益。

软件开发团队对软件的修改，即软件开发团队对软件的访问，也是软件利益

的一部分。为了避免不同的软件开发团队对软件的访问通道不相互冲突，也要进行软件的拆分。如上的三种情况也解释了这一点。

软件的利益也是开发团队利益的实现方式之一，因为软件是开发团队的产品，是为用户访问服务的。所以，用户的访问会直接关系到软件开发团队的利益。对软件的分工，使得每个软件减少了责任，只负责一部分的业务，每个软件的成长空间更大了，可以容纳更大的访问量。

18.4 软件开发的基础技术

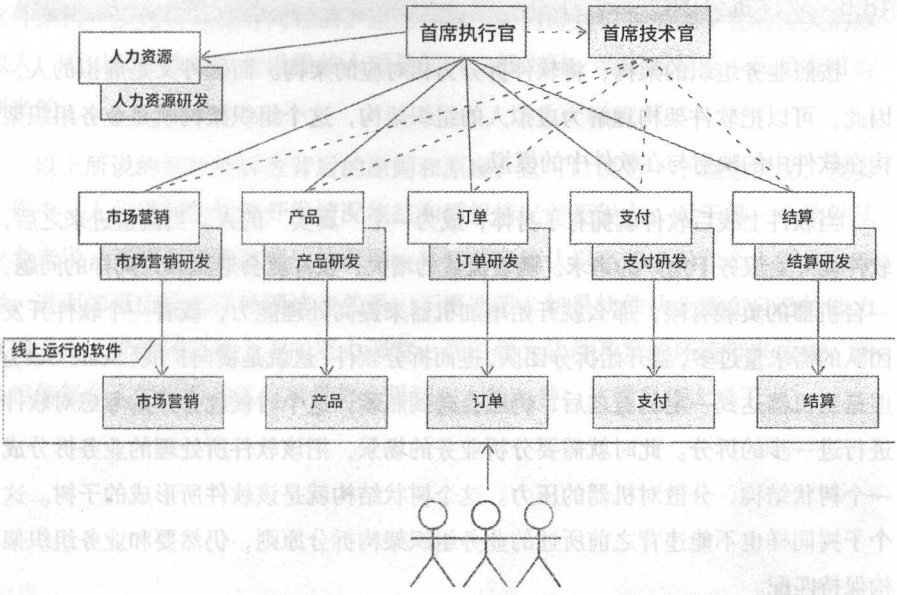
当软件开发进行到一定程度，每个部门的软件开发团队都会发现很多开发方面共通的东西。因为大家是在同一个公司，所开发的东西都放在同一个网站上，这时站在整个公司的层面，就会有很多大家都要共同遵守的东西出现。为了避免每个团队都重复发明轮子，这个时候，整个公司的软件开发团队也会产生统一的规则和相应的分工：

（1）同样都面临的技术问题，比方说 UED（User Experience Design）、日志、流量分流、基础框架、运维等，会专门有独立的团队来负责，这样开发团队就分成了专门服务于业务的软件开发团队和专门服务于技术的软件开发团队。

（2）如果是共享运维团队，那么还需要同样的开发流程保障，也会有独立的流程团队来服务于各软件开发团队。

所以业务软件开发团队就变成了这些技术软件开发团队的业务方，也需要相应的组织架构来处理业务软件开发团队提出的需求，也是一个树状结构。外部开源的工具，往往主要也由这个技术软件开发团队采纳并支持，为公司内部提供服务。技术软件开发团队开发出来的工具，往往也会对外部开源，为业界做出贡献。这样以技术为导向的同学，就知道自己是处在架构的什么位置，从而不会有那么多关于技术和业务关系的困惑。

对于 CEO 而言，需要一个 CTO 来辅助 CEO 管理软件方面的事务，管理软件开发团队的分工。对于业务软件开发团队所共同面临的问题，建立相应的内部分工，形成软件开发内部团队，配合业务软件开发团队进行工作。这样就形成了如下图所示的组织架构和软件架构：



以上就是业务组织架构的拆分，它决定了软件开发团队的组织架构，进而决定了软件的组织架构。业务组织架构拆分的影响非常深远。软件开发团队本身又进行了自己内部独特的分工，如分为业务和技术两类。技术是软件自身的业务体系，解决业务的模拟问题，以及如何让用户访问到达业务的模拟，即访问通道业务。软件则分解形成了不同的组件，落实到每个研发人员的身上，形成树状的关系。

换句话说，软件架构把软件看成虚拟的人，实际上就是虚拟人的组织架构。架构拆分的原则首先来源于业务自身的组织架构，使得软件架构保持和业务组织架构的匹配关系；其次来源于软件开发团队自身的组织架构；最后来源于用户的流量对软件本身的冲击。如果软件开发团队的组织架构和业务的组织架构一致，这就是损耗最小的方式，软件的架构也会更简单。

如果认识不到这么深远的影响来源，那么软件工程师和架构师在进行软件开发活动时，一定会遇到很多的麻烦：关系错综复杂、难以理清、失去控制。一旦从源头入手整理，一切就都清楚了。如果新接手一个团队，搞清楚这些也有利于快速展开工作，并可以在自己的范围内，尝试调整好这些关系。对于领导而言，认识并调整好这些关系，往往事半功倍。

18.5 软件拆分的第二动力

按照业务组织的架构，将软件拆分为相对应的架构。而软件又是虚拟的人，因此，可以把软件架构理解为虚拟人的组织架构，这个组织架构就是业务组织架构在软件中的映射与在软件中的模拟。

当软件上线后软件就拥有了身体，成为一个“真实”的人。当流量进来之后，软件就开始服务于用户的请求。随着流量的增大，软件就会遇到和人同样的问题，一台机器的负载有限，那么就开始增加机器来提高处理能力，或者一个软件开发团队的需求量过多，就开始拆分团队，进而拆分软件，这就是横向扩展（Scale Out）。但是当机器达到一定的量之后，仍然会遇到瓶颈，这个时候就会开始考虑对软件进行进一步的拆分。此时就需要分析业务的场景，把该软件所处理的业务拆分成一个树状结构，分散对机器的压力。这个树状结构就是该软件所形成的子树。这个子树同样也不能违背之前所述的业务组织架构拆分原则，仍然要和业务组织架构保持匹配。

综上来看，影响软件拆分的动力就是流量的增长，也就是希望单位时间内可以处理更多的业务。一方面流量增长导致了业务本身的拆分，进一步导致了软件的拆分；另一方面，某个软件本身流量的增长也会导致该软件自身的拆分。根本问题都在流量的增长上。从这个角度来看，流量增长是所有企业都追求的目标，只有更多的流量，业务才能够做大，良好的架构则是流量增长的保证。

18.6 架构一步到位

很多人会说这太麻烦了，可不可以把软件架构一步到位地做好？在当前的技术水平下，这基本上是不太可能的。一方面在流量不大的时候，做太复杂的拆分会浪费大量时间和不必要的成本；另一方面是因为流量是外在的因素，有经验的人可以做出部分预判，但也没有办法完全预测流量会在哪个地方增长。所以既没必要一步到位，也不可能一步到位。只能结合运维和运营，及时地探测到流量的增长，在流量达到瓶颈之前，根据业务的规律进行合理的拆分，帮助快速地应对增长。同样，不要把这叫作架构演化，因为此时业务并没有发生变化，只是流量增长，把这叫作软件的长大可能会更适合一点。

如果软件技术的发展能承担全球七十多亿人的访问，同时成本也很低，那么

软件架构一步到位是没有问题的。这时相当于人们得到了一个很便宜的巨大的虚拟人，可以轻松地应付全世界的人对他同时进行访问。我们相信这一天是有可能到来的。

以上所说的是软件拆分背后的原则和原理，这可以说是一个理想情况，现实工作中，人们遇到的软件开发情况往往和理想情况差距很大。对于每一个软件从业者来说，都需要理解这背后的原则，并能清晰地认识当前组织和个人所处的环境，进而知道应该怎样处理这些关系，不再迷茫。如果软件从业者在自己的能力范围之内，都尝试往这个方向努力调整，真正的为软件开发的环境带来改变，那么相信每个人的生活和工作质量都会得到极大的改善，这就善莫大焉了。

第 19 章　如何写好代码

代码是最容易写的，因为随便怎么写，只要符合语法就能够运行起来。代码也是最难写的，要想让代码容易维护，还要和业务一起长大，使得软件架构容易随着业务的长大做出新的拆分、合并，并要保证正确性。同时达到这么多的要求是非常困难的。借助之前讨论的软件架构，以下着重讨论一下代码应该如何写才能满足架构长大的要求，既容易维护，又能保证正确性。

软件生命周期中有两个最主要的子生命周期：软件开发生命周期和软件运行生命周期。其中，软件运行生命周期是核心生命周期，软件开发生命周期是为软件运行生命周期服务的，所以软件开发时必须要考虑到软件运行生命周期的特质，这一特质也决定了代码应该如何组织和编写。

在前面软件的定义中讨论过，软件实际上是对现实业务的模拟、虚拟化。结合软件的核心生命周期，也就是运行生命周期，可以看出软件主要是完成对业务生命周期的模拟，在此基础上完成用户的访问生命周期，使得用户能够到达模拟的业务生命周期。因此，代码主要由两部分组成：

（1）表达业务生命周期的代码。很多人把这部分叫作业务逻辑（Domain Logic），或者叫业务模型（Domain Model）。

这部分是来源于生活与业务，必须和现实生活及业务保持一致，忠实于现实中的业务。保持和现实生活一致的意思就是，业务代码的拆分，要和现实生活中业务人员的职责拆分一致，并保持内聚。也就是要根据业务的生命周期，分析出业务的核心生命周期，并识别出非核心生命周期，以及它们之间的组织方式，用代码表达出来。这些生命周期对应的负责角色同时也需要明确。

内聚，是指某个生命周期的变化是累积在一个生命周期主体上的，而不是分散在不同的主体上。在代码中表示一个生命周期主体，就是指一个对象。一个对象的名字，就是该生命周期主体的名字；一个对象的方法，就是该生命周期主体

的某个生命周期活动；一个对象的内部数据，就是该生命周期主体做生命周期活动的结果，累积在该对象上。某一时刻一个对象的内部数据，就是该生命周期主体当前的状态。对象的内部数据，会随着该生命周期主体的生命周期活动的进行以及时间而发生变化。对象内部数据的变化，既记录着业务的变化，也记录着生命周期活动的推进。

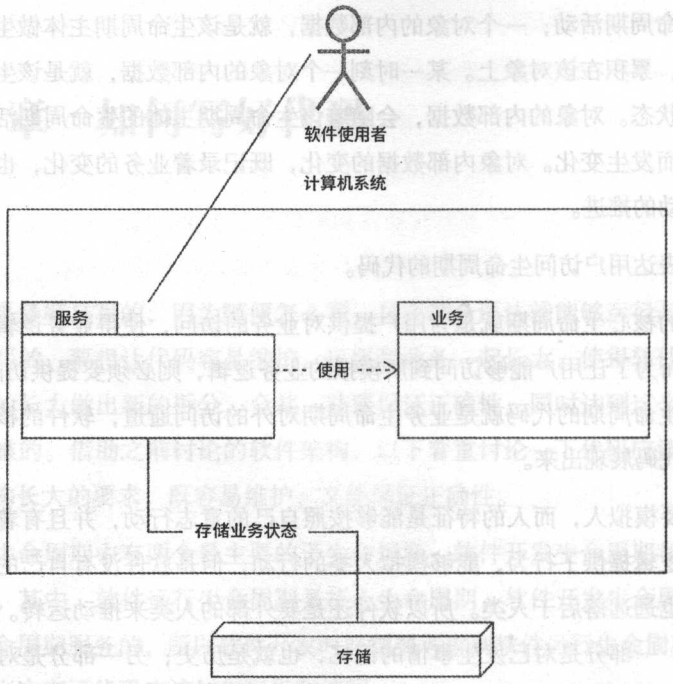
（2）表达用户访问生命周期的代码。

软件的核心生命周期就是为用户提供对业务的访问，使得业务逻辑能够服务于用户。而为了让用户能够访问到所模拟的业务逻辑，则必须要提供访问的通道。表达访问生命周期的代码就是业务生命周期对外的访问通道，软件的核心价值通过这部分代码展现出来。

软件要模拟人，而人的特征是能够按照自己的意志行动，并且有着自己的记忆。业务逻辑提供了行为，能够模拟人类的行动，但是软件没有自己的意志，在记忆层面也远远落后于人类。所以软件还是要外部的人类来推动运转。而记忆包括三部分：一部分是对已发生事情的记忆，也就是历史；另一部分是对当前行为结果的记忆，也就是当前状态，即将成为历史；还有一部分存储操作，也就是自己行为的规划，包括对未来的规划，也就是对行为的编程。

冯·诺伊曼的计算机体系架构把记忆和操作都实现了。只要理解了人类的特征，就不难理解冯·诺伊曼体系架构，为何要把存储和执行分离，同时把操作也数据化，这实际上都是为了模拟人。业务的操作编程，就是属于行为的记忆。但是已发生的历史和当前行为结果的记忆，由于计算机内存存储无法持久，所以需要把软件的状态通过存储来单独保存。当然，人也有记不住事情的时候，此时重要的内容就会写在外部存储中，比如以文档等方式存储在文件柜中。记忆保持在外部所带来的后果就是，每当需要加载状态时，都需要从外部存储中先获取。每当要修改状态时，都要及时保存到外部存储中，这一点无疑也是对现实生活中人类外部存储的模拟。

所以软件的代码可以分为三个部分，如下图所示。



要模拟一个完整的人，就需要业务（Business）部分来实现业务的逻辑，完成对业务生命周期的模拟。业务的状态要靠存储（Repository）来存储持久化，相当于现实生活中的文件柜。

那服务（Service）是用来做什么的呢？

首先，由于计算机和软件没有自己的意志，因此其内部生命周期的变化就要由外部的人来推动，这时需要提供一个访问通道给外部的用户。但如果把业务直接给用户访问，那么用户是很难和业务沟通的，因为这些专业术语，不是用户所能够理解的。

比如客户去银行，接待客户的是更接近用户语言的银行柜员，而不是银行内部的专业业务人员。柜员就是一个服务，以用户听得懂的方式和用户沟通，并把用户的要求转换为业务的语言，再由银行内部的专业业务人员执行相应的操作，柜员最后把执行的结果转换为用户的语言，为其服务。所以在这里服务提供的就是一个访问通道，为用户提供一个容易访问银行专业服务的访问方式。

另一方面，企业为了接待用户的访问，一定会设一个前台。客户对企业的访

问，统一由前台来接待。这样做的好处是，可以防止内部业务人员的工作被外部的访问随意打断，使得内部业务人员可以专心自己的本职工作。如果总是被外部访问打断，则内部人员的工作效率就会非常低。此时，这个前台就是一个服务。

最后，不同的用户访问，也要提供不同的服务，以避免不同的用户之间互相影响，因为服务的重用就是自找麻烦。比如银行的柜员，对 VIP 和普通用户就是不同的通道。更先进一点的银行，对操作烦琐的业务和操作简单的业务分开处理。如果存取钱和开户混在一起，那么就会让存取钱的人抓狂。ATM 机就是这个思路，让用户自行操作简单快速的业务。

服务作为一个通道的含义是什么呢？通道的意思就是只负责调用业务逻辑，但不包含业务逻辑。比如用户到银行存钱，柜员接到用户的钱后既不会把钱带回家，也不会给用户利息，用户同样也不会找柜员要本金和利息。柜员只是用户和银行之间的二传手，而不是银行业务逻辑的拥有者，这就是通道的意思。这也意味着软件工程师所写的服务代码中是不能够包含业务逻辑的。

从上图可以看出，服务为完成用户访问生命周期，承担的责任包括了组合业务和存储这两个，还要提供给用户访问。这部分的代码任务太多，代码人员的负担比较重，也容易引起服务的代码失控。为解决这个问题，需要把用户访问生命周期再展开分析一下。用户要完成访问业务逻辑生命周期，需要做如下事情：

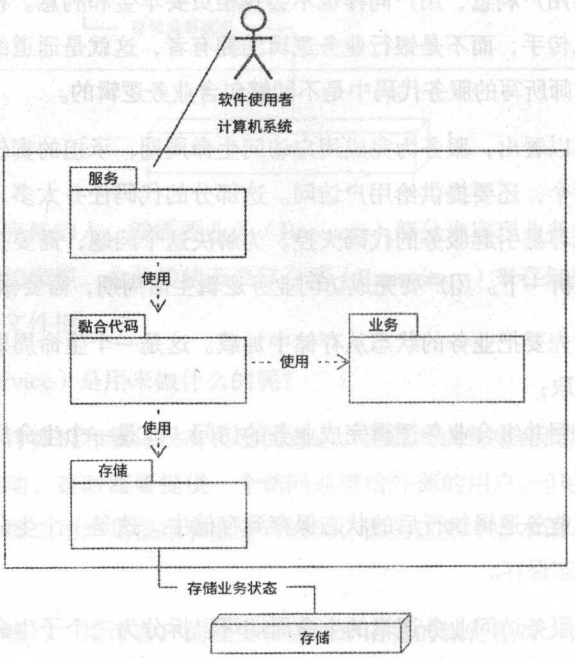
- 服务首先要把业务的状态从存储中加载。这是一个生命周期，主体为业务状态获取；
- 服务调用并组合业务逻辑完成业务的访问。这是一个生命周期，主体为业务访问；
- 服务把业务逻辑执行后的状态保存到存储中。这是一个生命周期，主体为业务状态保存。

也就是说服务访问业务逻辑的生命周期可以拆分为三个子生命周期。这三个子生命周期中，服务调用并组合业务逻辑完成业务访问是核心生命周期。所以服务还可以继续进行拆分，只保留组合业务逻辑即可，而状态保存则交给单独的组件，就形成了以核心生命周期为主轴，非核心生命周期围绕核心生命周期的树状架构，Service→Glue Code→Business 是核心生命周期，也就是树干。这样存储可以独立应对其变化，服务可以独立应对用户的需求变化，黏合代码（Glue Code）

负责从外部存储加载和保存业务逻辑的状态，把业务逻辑包装成具备记忆能力的虚拟人，供服务进行组合调用。

黏合代码是什么意思呢？业务逻辑属于行为是没有记忆的，而存储属于记忆是没有逻辑的。要把行为和记忆黏合在一起，才能够模拟一个人。因此黏合代码只有做到了这一点，对外才算是一个真正的虚拟人。服务作为通道把黏合代码和用户联系起来，成为了业务虚拟人和用户的桥梁。这和银行的柜员作用是一样的，名字和现实生活一样叫服务。这就是现实生活如何在软件中实现模拟的。

代码拆分之后，每一个部分都能够独立地变化。用户需求的变化、业务的变化和存储的变化，互相之间就不会产生连锁响应，并可以提升开发的并行度。如下图所示：



有些人会问，为何不把存储挂在服务上？这样黏合代码就不需要了。确实是可以的，大家把这个方式叫作事物脚本（Transactional Script）。但这样做的后果就好比让银行的核心业务人员直接接待客户，用户的访问因为沟通效率低就非常困难。黏合代码相当于是一个具备行为和记忆的完整业务人，不应该直接面对用户。

正确的方式应该是给用户独立的访问通道，就好比银行采用柜台人员来做接待一样，服务相当于柜台是面向用户的。用户的需求是变化最频繁的，服务的方式可以避免频繁的用户需求变化对内部分工的冲击。没有服务的保护会导致用户的需求直接冲击存储，而存储非常脆弱必须保护起来。

还有人会问，为何不把存储挂在业务上？这也是可以的，大家把这叫作活动记录（Active Record）。但这会让关心业务模型的代码人员，受到存储的影响，必然无法专注于业务生命周期上。并且存储的变动，会极大地影响业务生命周期，所以一般也不采用这种方式。毕竟内存持久化的问题是计算机体系结构本身的问题，不应该由业务代码人员来解决。

采用存储挂在黏合代码上的方式，可以让黏合代码成为一个完整的虚拟人，虚拟人具备记忆和行为，可以均衡地处理上述两个问题。有以上两种方式的好处，而又没有太大的弊病，这才是最合适的。因此代码就划分出了以下几个责任：

（1）服务专注于用户（User）的需求，通过组织黏合代码，也就是虚拟人所提供的生命周期活动完成需求。

（2）黏合代码专注于管理业务中对象的生命周期，并且通过存储保存或加载业务中对象的状态，实现对业务虚拟人的模拟。

（3）业务专注于实现业务的生命周期活动。

（4）存储专注于数据的保存和加载，并让数据和存储设备的存储粒度一一对应。

大家也可以发现，服务、黏合代码和存储的实现都需要计算机相关理论知识，作用主要是响应用户的需求，属于用户访问生命周期的架构分拆。而业务更多的是需要面对现实生活中的业务，其内部代码的架构分拆就直接反映业务的生命周期架构分拆。所以开发人员就可以依此进行分工，形成软件开发团队内部的组织架构。

分工后大家需要的知识储备可以减少些，软件工程师的进入门槛也因此降低了。只要不同部分的开发人员互相商量好接口定义，不同部分的开发就可以并行地进行，不仅提升了开发效率而且缩短了开发时间。也正因为有这样的内部分工，软件可以被拆分为不同的组件，形成以组件为单位的开发。

19.1 什么叫业务逻辑

根据以上的代码分工可以很清楚地看到，业务中的代码是最重要的，代码是系统的核心。服务与黏合代码中的代码起通道作用，这是为了让用户的请求能够访问到业务代码。因为真正干活的是业务代码，所以服务和黏合代码中的代码都是访问逻辑。而业务中描述的都是业务的生命周期活动，也就是人们通常所说的业务逻辑。

访问逻辑的特点是组合代码，即常见的顺序调用。这种代码里既不会有计算也不会有 if else 等判断，只有简单的组合代码，用于组合下一层的节点所提供的功能，方便上一层节点的调用。按照时间顺序执行是计算机的特性，由编译器来决定是计算机最擅长的。最本质的原因是计算机的基础都是图灵机。由此也可以想到，业务都是有核心生命周期的，生命周期也是按照时间顺序来执行的，这也正好符合图灵机的要求。

业务逻辑则是以业务核心生命周期为主线，切分出的一个树状架构。树上的每个节点都有其独有的生命周期，每个节点都有一个独立的概念来表达这个生命周期的特质。切分出来的生命周期是否完整决定了内聚性，大家讨论架构时总是提起内聚，但是很多人都不太理解什么叫内聚。

内聚就是要确保一个事物的生命周期是完整的，而不是分裂的。所谓完整，就是指一个生命周期的主体，从生到死之间的整个过程中，所发生的行为和状态是累积在一个主体上的。

如果生命周期中的某些行为和状态累积到了其他的生命周期主体上，这就不是内聚了，不内聚会产生权责不对等的问题。做好内聚的前提就是去发现生命周期及其主体。

从流程角度讲，访问逻辑实际上就是实现业务流程的基础。每次访问都通过对不同角色的生命周期，也就是不同的业务逻辑进行访问，访问其实就是遍历架构树，形成业务流程，完成用户生命周期活动的推进。

19.2 业务逻辑分散的危害

如果业务逻辑不是内聚的就会散落到很多其他地方去，比如散落到服务代码、

黏合代码或存储代码中，这会造成哪些问题呢？每个情况分别讨论一下：

（1）如果服务代码中混入了业务逻辑，则服务做了两件或者两件以上的事情。服务本身的责任是访问逻辑，这是顺序执行。加入了业务逻辑就表明做了两件或者两件以上的事情。可以分为以下两种情况：

a. 典型的情况就是两个不同的访问生命周期合并在一个服务中来实现。

比如两个不同类的用户共享一个服务方法，并在服务中判断区分这两类的访问分别处理，这就是人为造成的不必要逻辑。一旦某个用户访问生命周期发生变动，共享的另一个访问生命周期的执行就必定会受到影响。被影响的用户访问生命周期自己是不会跳出来说的，因为软件逻辑没有自己的意志，需要用户推动才可以。往往只会等到上线之后，最终用户推动访问生命周期活动时，发现受影响了才会显现出来，此时已经造成生产事故了。

如果有足够的责任感，也会主动去沟通所共享的其他生命周期是否受影响。可是其他共享访问的责任方并没有动力来配合，何况有可能还会给自己增加工作量。这就形成了权责不对等，导致发现问题的沟通成本非常高，因此最后都是不了了之，等待生产事故出现来教训大家。最可惜的是即便引发了生产事故，责任方还是意识不到这个问题是因为共享了访问通道而造成的，导致类似的问题一而再再而三地发生，修改代码也变得战战兢兢，大家都疲惫不堪。

这就是为什么不同用户的访问通道一定要隔离，不能重用，不能互相影响。必须把这个服务代码拆分，把不同访问生命周期主体的访问通道分离，确保每个服务只做一件事情，只为一类用户提供通道，确保不同用户的访问生命周期之间没有共享，保证用户访问生命周期本身的内聚，确保不同类型用户的访问通道是独立的。

拆分之后，用户端就要自行组合不同的服务来完成自己的执行流程。如果不拆分的话上线后会出现很多不可预料的问题，最终会因为用户的利益受损而返工，从而也损害了自己的利益。很多软件工程师上线时会没有信心，大部分原因就在此。

b. 如果是有计算的逻辑的话，比如受益计算、订单金额计算等，那么这部分应该是业务代码需要完成的，不能交给服务代码来实现。这部分代码是需要单元测试的，而服务代码要和用户打交道就会有上下文相关的代码，因此不适合做单

元测试。

（2）黏合代码里面如果包含业务逻辑的话，也会做两件或者两件以上的事情，会和服务代码一样，遇到同样的问题。

（3）存储代码里面如果混入了业务逻辑，则会导致业务逻辑进入到存储设备中。

编写存储访问代码的语言常见的是 SQL（Structural Query Language）。如果业务逻辑混入 SQL，会导致 SQL 写得很复杂，里面会做很多判断和计算，导致存储 CPU 不堪重负。

照理来说存储的 CPU 应该是很闲的，因为存储设备的主要目的是存储，而不是计算。逻辑计算不是存储的强项，存储代码做逻辑编程的难度也非常高。并且这些业务逻辑往往和所存储的数据直接相关，只能够做本地计算。存储一旦变成了逻辑计算的主体，绑定数据的逻辑计算就成了一个巨大的限制，会导致存储设备无法通过增加机器的方式横向扩展长大，只能换性能更好的机器纵向扩展（Scale up），而纵向扩展不仅程度有限而且成本也很高。

通过以上的分析可以看到，业务逻辑如果不内聚的话，它分散到哪里，哪里就会受到限制，哪里就有麻烦，哪里就会冒火。并且最后都会导致架构无法快速得横向扩展和拆分，增加修改的难度、不确定性和成本。这些都不符合开发人员以及业务的利益。

19.3 业务逻辑内聚的好处

那么保证业务逻辑的内聚性，好处有哪些呢？

（1）由于服务、黏合代码和存储里面的代码都是严格的访问代码，而访问代码的特征就是简单的顺序调用，因此这些代码只要做连通性测试即可。只要保证访问都是连通的就没有问题，不需要单元测试。这是真正的组合。

另外因为这些代码都需要和很多环境相关的上下文打交道来交换数据，所以很难做单元测试。很多人讨论单元测试必谈模仿（Mock），实际上就是因为逻辑散落到访问代码中去了。而访问代码总是需要和上下文打交道，通过交换数据来达到通道的目的。

如果访问代码要做单元测试的话，就需要把上下文模拟出来，才能够获取到

数据，并执行代码。因此模仿代码就是要模拟出这些上下文来，如服务器相关的上下文、存储相关上下文、缓存相关上下文和网络相关上下文等。模仿的维护成本慢慢会变得越来越高，导致到最后单元测试形同虚设。很多人还因此得出一个结论：单元测试是骗人的！可惜可叹。

（2）由于业务不访问任何计算机设备相关的上下文，不访问任何具体的设备，如数据库、缓存、中间件等，因为其状态是靠黏合代码来填充的，因此这部分代码是非常容易写单元测试的，而且单元测试必须保证 100% 覆盖。

由于除业务代码之外，其他地方的代码没有业务逻辑，所以一旦系统有问题，马上可以断定要么是业务逻辑的问题，要么是访问生命周期中的环境问题。访问的问题很容易通过监控发现，业务逻辑的问题可以通过单元测试发现。如果单元测试没能发现这个问题，那么单元测试还需要进一步的补充场景。因此每一次的问题发现和修复都让代码变得更加强壮，测试场景也会更完整。同时线上问题的模拟也就变得非常简单，大家不用再加班加点地紧急处理线上问题。

（3）存储如果没有逻辑，则很容易按照存储设备本身的最小访问粒度来完成工作。比如关系型数据库（DataBase，DB）就完全可以做到单表访问。由于存储设备只关心存取数据且和业务没有关系，做表的拆分也是非常容易的事情，所以存储设备通过增加机器就可以横向扩展长大。

很多人担心如果没有了 Join，访问数据库的次数是不是更多了？性能是不是会下降？按照现在的网络条件，网络访问（Network IO）和磁盘访问（Disk IO）的差距已经不大了。在合理的设计下，多访问几次数据库并不会引起这个问题。另外如果有多台数据库的话，还能通过并行加速访问。

数据库横向扩展的另一个大问题是数据库拆分之后的事务问题，这个问题是大家非常关心的，我们留待后续重点讨论。

（4）服务代码、黏合代码和存储代码只有变简单了，才可以让开发人员投入更多时间来研究业务。毕竟模拟业务才是软件所真正服务的对象。

19.4　代码架构实例

再来看一个实际的例子以加深理解，如下图所示。管理者（Manager）类实际

就是黏合代码。有几个注意点需要说明一下：

（1）不能把业务模型（Business Model）当作数据对象来处理。

业务模型关心的是其生命周期，数据是这些生命周期行为的状态。所以黏合代码需要把业务模型转换为存储设备的实体（Entity），实体和存储设备里面的存储粒度一一对应。比如在数据库中，每个实体对应一张表，并且跟着表的变化而变化，这样就保证存储设备的变更不会影响业务模型。同样业务模型不能拿来用作服务和用户之间的数据交互媒介，只能转换为 DTO（Data Transfer Object）来使用。也就是说业务模型对用户是不可见的。DTO 的目的是在用户的访问操作中传输数据，并和用户交互的视图（View）保持一致。通过增加 DTO 可以保证用户的访问生命周期需求变化，并不会影响到业务模型，虽然用户的访问生命周期需求变化是最频繁的，但 DTO 解决了这个难题。

（2）服务代码里不要考虑代码重用。

针对不同的角色要提供不同的在服务来服务，确保他们之间的访问生命周期是隔离的，避免互相影响。如果多个不同的角色访问同一个接口，一旦其中某个角色的需求发生了变化，就会要求开发人员去修改，而这个修改总是会影响到其他的角色，人为地造成权责不对等。

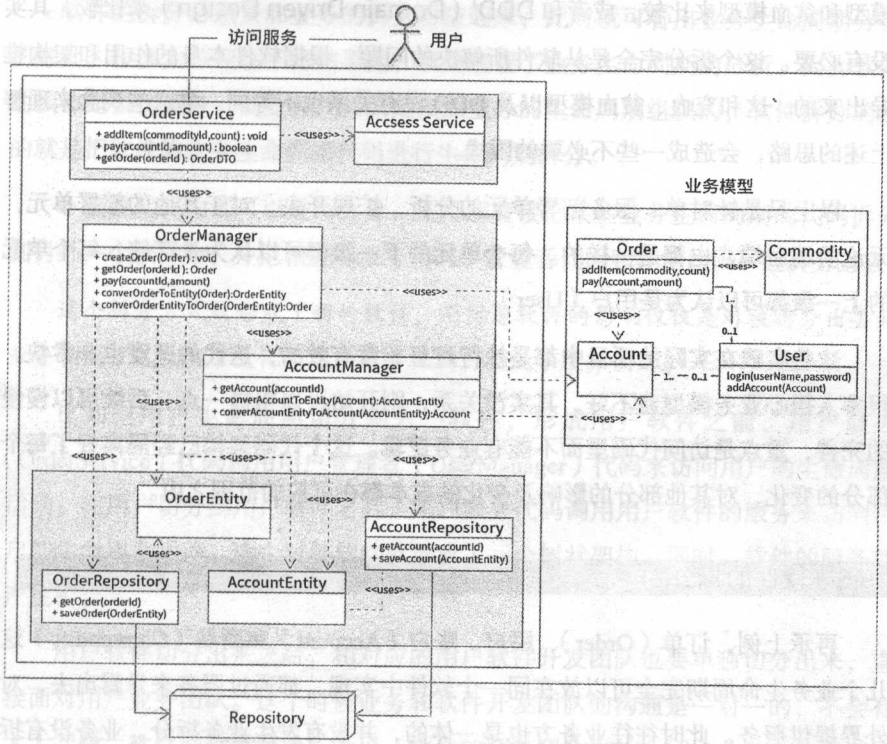
共享的角色越多，影响的范围就越大。由于修改前需要这些角色一起配合确定是否受影响，但是受影响的这些角色因为没有需求往往会敷衍了事。这样很多不必要的沟通就产生了，产生的成本非常之高，最终都会导致发布的事故，从而造成线上的问题影响最终用户。所以尽量给不同的角色不同的服务，既避免通道重用又降低沟通成本。

但是服务太多不就有问题了吗？

服务多不是问题，服务的生命周期管理才是问题。服务治理（Service Governance）中心就是来解决这个问题的，这时服务注册和查找等需求就出现了。因为服务里面没有逻辑，所以开发和管理就非常简单，可以快速应对业务的变化。软件工程师们只有更快地变，更容易地变，才能更好地应对变。

（3）业务模型是必须要重用的，因为这是所有用户访问的目标。所有服务代码需要的，就是组合业务模型所代表的业务生命周期，用来完成用户的访问。业

务模型总是围绕着核心生命周期展开的一个树状架构。如果不是，就要看看业务核心生命周期的识别是不是出了问题。



19.5　代码误解

服务代码、黏合代码和存储代码不能有逻辑，很多软件工程师在实际的操作中非常不理解。要么认为这个根本做不到，要么认为增大了工作量。

做到这一点确实需要很多的学习成本，特别是要克服对业务的恐惧。我的游泳教练曾和我说过这些话，令我至今都记忆犹新，大意如下：“业余选手要确保呼吸，总想把头抬起来，身体反而沉下去。只有克服呼吸进水的恐惧，把头往水里压下去，身体才能够从水里浮起来，这样才能真正确保呼吸。真正专业的习惯往往与人们日常的反应相反”。如果真正想快速地完成代码工作，就要克服自己对时间的恐惧，真正的去研究业务的核心生命周期，研究相关利益人的利益，把这个变成日常的习惯。写代码的时候让该出现逻辑的地方出现逻辑，让不该出现的地

方不要出现。而一旦不该出现的地方出现了逻辑，就要马上采取行动纠正过来。

很多人可能会把这种做法和马丁·福勒（Martin Fowler）曾经提出过的充血模型和贫血模型来比较，或者和 DDD[1]（Domain Driven Design）来比较，其实没有必要。这个拆分完全是从软件所解决的问题，根据软件本身的作用和架构推导出来的，这和充血、贫血模型以及 DDD 没有关系也不等同。拿这些观念来理解上述的思路，会造成一些不必要的困扰。

以上只是针对单一服务部署单元的分析，扩展开去，对于其他的部署单元，无论前端后端，也都是一样的。每个单元的下一级都可以认为是存储，每个单元的上一级都可以认为是用户（User）。

这些实践在实际的项目中都是执行过且非常有效的，迭代的速度也非常快。很多人担心业务模型建不好，其实没关系，刚开始可以粗糙一点，后续可以慢慢的完善，重点是访问代码里面不能有业务逻辑。这个代码架构已经隔离好了每个部分的变化，对其他部分的影响及变化的成本都在可控的范围之内。

19.6 软件的拆分

再承上例，订单（Order）、用户、账户（Account）和商品（Commodity）这几个业务生命周期完全可以放在同一个软件中实现，都通过服务来暴露出去，对外界提供服务。此时往往业务方也是一体的，并没有发生业务拆分。业务没有拆分，为何这些生命周期却拆分出来了呢？这些生命周期的拆分其实是人类有历史以来经历分工形成的行之有效的组织方式，即模式。每个企业成立后，都可以利用这些拆分的方式来组织自己的业务。

当业务逐渐的长大，为了提高效率，用户、账户和商品会逐渐地切分出来，形成独立的生命周期，变成不同的部门单独负责。也就是说业务部门会发生架构的切分。此时几个部门同时来对一个软件提需求，部门之间的沟通就变成了问题，软件发布上线排队也成了问题，软件的开发生命周期的效率需要提升。另一方面，随着用户、账户和商品的增长，订单也会逐渐增长，而软件越来越庞大，一个软件的运行效率也成了问题，软件的运行生命周期的效率需要提升。因此软件必须

1 DDD（Domain Driven Design），领域驱动设计。

要进行拆分才能够提高软件的开发生命周期的效率，同时软件的拆分也会导致软件开发部门的拆分。

软件的拆分必须要和业务的拆分对应起来，此时就可看出业务生命周期分析的好处。很多人总是拆不好的原因，就是忽视了业务生命周期的分析。因为软件的核心是模拟业务，而业务代码又是按照业务的生命周期组织的，软件拆分的目的就是把软件的业务生命周期代码进行生命周期拆分。

不仅仅代码内部可以进行拆分，还可以直接把某些业务生命周期的代码拆分到另外一个软件中，并把相应的服务代码、管理者代码和存储代码一起拆分过去。

这个拆分方式就形成了新的软件，而对原软件的影响仅仅是对被拆分出去的业务调用方式发生了变化而已，从本地调用变成了服务调用。

比如把用户生命周期切分到另一软件，形成用户软件之前，用户服务（OrderService）代码调用用户管理者（UserManager）代码来访问用户的生命周期活动；在用户切分到用户软件之后，用户服务代码调用用户软件的服务来访问用户的生命周期活动。这个时候软件就形成了一个树状架构。同时，软件的服务就产生了。

用户软件切分出来之后，相对应的用户软件开发团队也要单独切分出来，直接面对用户业务团队。这个时候业务和软件开发团队的沟通是一对一的，不会有太大问题。软件因此就开始长大了，软件的开发团队也长大了，业务、软件开发团队和软件的效率都得到了提升。

从以上的软件拆分可以看出，业务生命周期的分析既是软件拆分的大前提又是架构的基因。失去了这一点，软件架构就会出现问题，软件开发团队的组织架构也会出现问题，会导致产生大量不必要的额外沟通来弥补。此时软件开发团队的组织就像人一样，人的身体失衡就开始生病了。一旦分析清楚了业务的生命周期，软件的运行架构和软件开发的组织架构可以很容易地进行拆分调整，软件和软件开发团队都可以顺利的拆分长大。

第 20 章　单元测试

很多软件工程师对单元测试（Unit Test）非常的失望，甚至认为单元测试是骗人的。毫无疑问他们遇到了很多的困难，许多疑问也得不到解答。本章针对这些困难做些分析，希望对软件工程师有所帮助。毕竟架构的落地还是要靠他们的。

20.1　什么是单元测试

要认识单元测试，首先要明白什么是“单元”（Unit）。所谓“单元”指的是代码调用的最小单位，实际上指的就是一个功能块（Function）或者方法（Method）。当然不同的语言有不同的叫法，这里就不一一列举了。所以单元测试指的就是对这些代码调用单元的测试。

单元测试是一种白盒测试，就是必须要对单元的代码细节很清楚才能做的测试。所以，单元测试的编写和执行都是由软件工程师来做的。相对于单元测试，还有集成测试。集成测试基本都是黑盒测试，主要由测试人员根据软件的功能手册来进行测试，需要有专门的测试环境配合。集成测试又分功能测试、回归测试等，测试部分也是软件一个独特的子业务体系，这里就不一一展开了。

20.2　单元测试的困境

有人会问，如果方法就是单元，一个软件包含那么多方法，那单元测试岂不是要写死人？确实会存在这个问题，这也是为什么很多人说单元测试根本不可行，甚至无法推广的原因之一。

另一个原因是，一提单元测试就有人提出来说必须要做模拟（Mock）。因为要把代码跑起来就必须要模拟上下文，这样代码才可以跑起来测试。

这么多单元要测，并且跑起来还要那么多外部环境要模拟，光是维护这么多测试就要比开发量还要大了，这不是找罪受嘛？更不要说单元测试也是代码，单

元测试本身是不是也要测啊？因此很多软件工程师反感单元测试，即便有强制规定也是敷衍了事，最后不了了之。

这就是为什么经过了这么多年的宣传，单元测试在软件工程师中仍然得不到普及的原因。既然开发人员无法验证自己的代码是否正确，那么测试这部分工作就产生了分工，不再由软件工程师来执行，而是在代码完成之后，把代码编译部署到测试环境，交由测试人员来完成集成测试。这就形成了一个架构拆分，形成了新测试的生命周期，由测试人员来推进测试的生命周期。

至今为止，大部分软件开发团队仍然完全依赖于人工的集成测试来提升软件质量。而集成测试属于新的分工，需要大量的人工介入，需要比较长的时间周期才能完成测试，也增加了很大的沟通成本。如果软件不经常进行迭代，人工测试问题还不太大，毕竟难得测一次。但是现代的软件开发迭代速度非常快，经常是按周的频率进行迭代。如果每周都要全面的测试，那就要维护一个比开发团队大得多的测试团队，这就有很大的问题了。测试因此就成为了迭代速度的瓶颈，会导致迭代速度的减慢，损害软件的长大以及业务的增长。

为了提升集成测试的效率，集成测试也可以进行大量的自动化。比如回归测试就是自动化的重点，但是在这里就不展开了。另一个提升测试效率的办法，就是让测试回归到软件开发工程师职责范围内，也就是单元测试。当然，单元测试也属于自动化测试。

为了让测试工作回归到软件工程师，还是需要开发人员掌握单元测试，自行开展测试的工作。针对大家对单元测试的误解，以下把单元测试的误区一一进行剖析，给出解决办法，供大家参考。

20.3　单元测试测什么

一个软件包含那么多的方法，单元测试怎么写得过来呢？虽然“单元”指的是一个方法，但并不是所有的“单元”都需要单元测试。既然要做单元测试，就要知道要测什么内容。比如如下的代码，需要测试吗？

```
public OrderDTO getUserOrder(HttpRequest request){
        String userId = request.getParameter("userId");
        String orderId = request.getParameter("orderId");
        UserDTO user = userManager.getUser(userId);
```

```
        OrderDTO order = orderManager.getOrder(user, orderId);
        order.setUser(user);
        return order;
    }
```

这是一个典型的服务代码，里面只有简单的顺序调用。假设要单元测试的话，究竟是测试什么的？测试编译器是否编译正确？所在的容器是否运行正常？测试CPU是否顺序执行？这些需要单元测试吗？

实际上简单的顺序执行是不需要调用的，只要编译正确，操作系统会保证它的正常执行。如果这段代码不正常了，绝对不是这段代码自身的问题。也就是说这样的代码是不需要写单元测试的。这类代码属于访问代码，也叫组合代码，本身没有业务逻辑。

既然这种代码不需要测试，那么如下的代码需要测试吗？

```
public OrderDTO getOrder(String userId, String orderId) {
    Order order = orderRepository.getOrder(orderId);
    User user = userRepository.getUser(userId);
    order.setUser(user);
    OrderDTO orderDTO = convertToDTO(order);
    return orderDTO;
}
```

这段是典型的管理者方法。按照前面的分析，这段代码是简单的顺序执行，也是不需要单元测试的。有人会问，可是里面有数据库访问，是不是需要测呢？如果要测，那么测的又是什么呢？是来测试数据库是否工作正常的吗？这似乎是数据库厂商的事吧。

这个方法确实不用单元测试，因为没有理由要测。事实上，服务代码、管理者代码和存储代码都是不需要写单元测试的。单元测试是用来测软件工程师自己写的逻辑，如果代码里面没有逻辑就不需要写单元测试。

上文其实也回答了为什么单元测试本身不需要测试，因为单元测试的代码都是简单的顺序执行，并没有逻辑。如果单元测试写得很复杂，比如还有判断、计算等，那么单元测试的代码本身就需要测试，这就不能叫单元测试了。

20.4 如何改造代码

但是开发人员们总是把服务代码写成这样：

```
    public OrderDTO getUserOrder(HttpRequest request){
        String userId = request.getParameter("userId");
        String orderId = request.getParameter("orderId");
        UserDTO user = userManager.getUser(userId);
        OrderDTO order = orderManager.getOrder(orderId);
        if (order != null && order.getUserId != null &&
order.getUserId.equals(userId)){
            order.setUser(user);
            return order;
        }
        return null;
    }
```

从代码里可以看到服务代码有了逻辑判断，因此这个代码就需要测试了。可是这段代码里面有容器的依赖 HttpRequest，就必须要模拟出来才可以测试，否则只有把服务器启动才能够运行这段代码。这就不是单元测试了，因为服务器启动起来就变成了集成测试。此时模拟代码就出现了，光是模拟容器还不行，还需要模拟数据库才能够跑起来，于是模拟就越来越复杂了。

只要出现了模拟，单元测试就开始失效了。

服务里面的这段逻辑，当时写起来是很方便的，但是给单元测试带来了太大的成本。可是这段代码又确实是要测的，怎么办呢？

需要单元测试的代码实际上是开发人员自己写的逻辑，测试逻辑所依赖的环境是否正常不是单元测试的目的。在环境访问代码中引入逻辑，只会让逻辑更难测试，导致逻辑代码无法进行单元测试。因此，可单元测试的代码，才能够采用单元测试。判断可测试的代码还有一个方法，就是看这个方法能否用一个 main 函数直接运行，如果可以的话就是可单元测试的代码。可测试的代码还有另一个特征，就是该方法单元的参数，开发人员可以自由模拟，不需要依赖外部环境。

如果代码里有逻辑，但是不可单元测试的话，就需要改造代码。改造的办法，就是要确保逻辑代码和外部环境相关代码隔离，这个逻辑代码就是可单元测试的。而隔离的办法就是把代码的执行顺序，也就是单元的执行生命周期，做架构的拆分。访问代码的核心就在于传递上下文的数据，因此先把逻辑需要的数据从上下文环境中取出来，给逻辑代码单独另建一个单元，再把从环境中取出来的参数作为入参传入新建的逻辑代码单元。这样新建的逻辑代码单元就只依赖入参，而不再依赖环境了，开发人员就可以自由地模拟输入参数做单元测试。环境相关的代码在

架构拆分后不再包含逻辑，恢复了简单的顺序执行，也不再需要单元测试了。如对前面不可测试服务代码的改造如下：

```
    public OrderDTO getUserOrder(HttpRequest request){
            String userId = request.getParameter("userId");
            String orderId = request.getParameter("orderId");
            UserDTO user = UserManager.getUser(userId);
            OrderDTO order = orderManager.getOrder(orderId);
            return checkUser(order, user);
        }
        public OrderDTO checkUser(OrderDTO order, UserDTO user,
String userId){
            if (order != null && order.getUserId() != null &&
order.getUserId().equals(userId)){
                order.setUser(user);
                return order;
            }
            return null;
        }
```

新建一个 checkUser 方法，把 checkUser 方法所需的 userId 参数从 HttpRequest 上下文中取出来的。这一改变，让 getUserOrder 方法不再需要单元测试，因为变成简单的顺序执行，就没有逻辑需要测试。而需要测试的 checkUser 方法变成和环境上下文无关了，此时可以直接用 main 函数来运行，这是能做单元测试的。

对于一个逻辑“单元”，也就是方法，所依赖的无非是两个方面：一个是输入的参数，另一个是内部对外部的方法调用。只要确保输入参数不包含外部环境的上下文，同时内部代码对外部的调用也不包含对环境上下文的访问，这个方法就是可以单元测试的。

所以，当软件工程师需要测试一个方法中的逻辑，但是发现该方法无法测试的时候，先不要急着去模拟，去改造这个方法再把逻辑剥离出来变成可测试的。这是最快的方法，也是成本最低的方法。

另外也可以看出，checkUser 实际上是订单的一段逻辑，分散到了服务中。最好把这段逻辑归还到订单，即真正的业务逻辑中。所以前述逻辑只应该存在于业务代码中的论断再次得到了验证。服务代码、管理者代码和存储代码中是不应该存在业务逻辑的，如果发现有逻辑，就可以用上述的办法剥离出来，归还给业务代码。

因此要写单元测试就要先搞清楚两个问题：

首先，要明确的是什么样的代码需要单元测试，也就是说要测的是什么。只有自己写的逻辑才需要测，外部环境和别人的代码都不是单元测试的目的，这个问题必须要先识别清楚。

其次，逻辑能否用单元测试来测。也就是说，自己写的逻辑单元是否可测。如果自己写的逻辑不能拿单元测试来测，或者需要用模拟才能测，说明自己写的代码不可单元测试，也说明逻辑和访问代码混合了。必须要先把自己的逻辑从访问代码中拆分出来，把自己写的逻辑独立出来，让自己写的逻辑单元只依赖于输入参数，就变成可测的了。能够通过 main 函数提供输入参数就可以运行起来的逻辑单元，都是可测的。

所以单元测试是一个很好的办法，把它交给软件工程师来编写，本身是没有问题的。有问题的其实是软件工程师对单元测试的认识，以及软件工程师的代码本身。因此有必要对软件工程师进行培训，让软件工程师知道什么样的代码才是需要测试的，如何让逻辑单元可测。一旦发现代码需要单元测试，但是必须要模拟才能进行单元测试的时候，就要让软件工程师意识到这是一个坏味道（Bad Smell），要赶紧把代码改正过来，并自觉养成这样的习惯。

20.5　为什么要做单元测试

很多软件工程师写了一段时间的代码后就觉得这个工作很无聊，因为没能及时得到自己所做工作的反馈，这会极大地影响软件工程师的成长。做任何事情并得到及时的反馈，这是形成兴趣爱好的必经途径。

通过别人的反馈保持的兴趣会比较短，因为很多时候别人的反馈都是出于礼貌。而自己对自己所做事情的反馈，才是至关重要的，这是形成兴趣的最重要因素。

如果每一步的努力，自己都能看到效果和进步，自然而然就会对自己产生正向激励，形成兴趣。这其实也是权责对等的意义，一定要让每个个体从自己的工作结果中获益，才能继续推进生命周期运作。

另一方面，对于软件工程师来讲，如果写代码时对自己写的代码没有办法快

速的验证，也就没有一个反馈，往往会有一种强烈的不安全感。写的代码越多，不安全感累积的会越多，最后会发觉自己对自己所写的代码完全没有把握，这是非常影响生活质量的。

即便是快速的迭代方式，最少也要一周才能够得到测试的反馈。并且很有可能测试的反馈结果会导致自己一周的代码都白写了，全部要推翻重来。所以测试人员在测试的时候，软件工程师非常焦虑。如果迭代时间更长的话，造成的心理压力会更大。测试在进行的时候，软件工程师往往会疲于奔命地去修复问题，也容易和测试团队发生冲突，从而产生沟通问题。

有了单元测试，每天能够看到自己的单元测试运行结果全部是绿色，或者正在逐渐把红色的变成绿色，心里的成就感是非常强的。由于形成了代码编写的反馈环，所写的每个代码都能够快速的从单元测试中看到变化，软件工程师会对写代码保持极大的兴趣，每天都能够有强大的动力去修改代码，而且心里也会感到更安全。在与人沟通时，心里也会更有底气。

并且有了单元测试，持续集成才是可能的。每天下班前做代码检入（Checkin），第二天早上上班来看自己的单元测试报告，这样每天对自己的代码都是心里有底的。这就建立了一个良性的反馈，提升了软件工程师的生活质量和工作质量，同时软件工程师的成长也就有了保证，对写代码也会培养出浓厚的兴趣。

另外，保障自己代码的正确性本身就是软件工程师自己的事情，不是测试人员的事情。只是软件工程师没法做到这一点，不得已引入测试人员来帮助做好这一点罢了。如果软件工程师做好了单元测试，并且掌握了单元测试技术，往往就不再需要有单独的测试团队了。单元测试一旦写好可以长期使用，特别是在回归的时候，可以帮助节省大量的测试时间，并提升质量。比如软件工程师修改了某些代码，并不一定能够估计到影响的范围，但是单元测试会忠实地反应影响的范围，可以帮助发现很多隐藏的问题。

单元测试不需要加载环境上下文，跑一次全量测试非常快，写完代码马上就可以得到反馈。而集成测试往往周期非常长，一次全量经常需要一天以上。有了单元测试，编写代码才可以快速响应业务的需求。

而且单元测试可以帮助识别业务逻辑是否分散到访问代码中。如分散到了服务代码、管理者代码或存储代码中，单元测试就不可行了，软件工程师自然就会

去改正。做好了单元测试，基本上业务逻辑代码就自然会和访问代码分离，形成代码的架构。

20.6　如何做单元测试

明白了测什么，怎么测就很简单了：通过提供预期的输入和预期的结果，与单元的实际运行结果进行对比，就可以知道单元的工作是否和预期的一致。各种输入参数的边界条件都需要测到。单元测试的覆盖率可以帮助识别单元测试是否存在没有运行到的代码。

以测试一个目标方法为例，基本上分为三个步骤：

- 构建输入参数，并预测该输入所产生的输出；
- 调用要测试的目标方法，获取输出；
- 检测目标方法的输出是否和预期的输出一致（Assert）。

对同一个目标方法，通过构建各种不同的输入，重复上述的步骤，检测各种正常与边界状况和预期的是否相符，确保把目标方法的各种可能性都覆盖。通过代码覆盖率的跟踪，还可以看到哪些代码分支没有走到，进一步丰富测试的案例。业务逻辑代码的单元测试覆盖率应该要达到 100%。

对于其他要测试的方法，同样重复如上的过程。单元测试本来也是一个生命周期，所以其产生的结果反映的就是逻辑代码修改的进程。

测试代码是对业务代码的检测，因此测试代码和业务代码的代码组织架构要保持一致，都要是树状架构，且与业务代码架构一一对应。以 Java 为例，每个要测试的 Java 类，对应有一个单元测试 Java 类，区别为单元测试 Java 类名后加 Test。要测试的 Java 类中每个要测试的方法，在对应的单元测试 Java 类中有一个对应的测试方法，在原方法名前加一个 test，方便测试出错的时候找到被测类和被测方法。单元测试 Java 类和要测试的 Java 类处于同一个包名下，方便访问 protected 类型的方法。

比如，Order 类会有一个对应的 OrderTest 单元测试类，都处在相同的包名下；Order. checkUser()对应的测试方法为 OrderTest. testCheckUser()；在 OrderTest. testCheckUser()方法中为 Order.checkUser()模拟各种输入条件，并检测是否是预期

的输出，来判断 Order.checkUser()中的逻辑是否正确。

为了避免测试代码打包到生产代码中，一般会把测试代码存放在另一个目录下，避免和生产代码混在一起，方便测试代码的管理和维护。比如上例中的生产代码会放在 src/main/java 路径下，测试代码会放在 src/test/java 路径下。

单元测试代码刚开始写确实是有一定的代码量，会觉得有点占用时间。但是随着代码的深入，测试工作量会愈来愈小，因为写过了的测试就不用再写了，直接运行即可。反观手工测试，刚开始确实很快，因为需要测的内容少。随着代码增多，手工测试工作量加大，反而花得时间更多，因为以前测过的内容还需要重新再测，无法把自己的重复工作自动化。

所以一旦养成习惯后就离不开单元测试了，因为会上瘾。一旦代码没写单元测试会觉得不靠谱，心里也不安。大部分时候，单元测试写完了并且覆盖率 100%，只要业务没有太大的变化，单元测试是不需要维护的。一次投入会持续受益，单元测试总是在那里给软件工程师保驾护航，给人极大的安全感。

第 21 章　软件架构和面向对象

面向对象是一个非常有争议的话题。而且大部分人讨论的面向对象其实是面向对象编程语言的特性，是和面向对象相对立的面向过程编程语言的一种改进。可是面向对象和面向过程不仅仅是编程语言，还是两种不同的思维方式。这两种不同的思维方式也不是相互对立的，就像一个硬币由正反两面组成一样，是同一个事物的两面，或称为“一体两面”，缺一不可。

21.1　什么是面向过程

面向过程（Procedural Oriented），也叫过程式编程（Procedural Programming），是计算机软件编程行业的术语。面向过程的编程方式，会把过程式的代码封装成代码块，也叫功能（Function）或过程（Procedure），通过功能调用（Procedure Call）或者过程调用（Function Call）来组织调用的流程。

为什么会形成这种方式呢？因为计算机软件是基于图灵机上的，而图灵机描述的就是顺序执行。毫无疑问，最早的编程语言都是顺序执行的语言，即便现在，所有的编程语言也都是顺序执行的。人类的语言也是过程式的，说的人必须要一个字一个字地说出来，听的人也要一个字一个字地连续听，才能够理解说的人所说的话。编程语言模仿的就是人类的语言。

再往更深的层次去说，世界上所有的事物都是随着时间变化而逐步发展变化的，即生命周期的规律。也就是说，面向过程表述的实际上是生命周期按时间推进，按顺序发展变化的特性。这是这个世界运作的规律。

在这种思考模式下，对业务的分析主要集中在业务流程的分析，把业务流程过程中的节点封装成功能或过程，然后再组织这些功能或过程，来完成业务流程的实现。

由于面向过程有这样的特性，因此越靠近硬件，越会使用面向过程的语言来

编写，比如操作系统底层、硬件驱动等。因为硬件是天然的顺序执行，必须采用流程式的思维，才能更好地配合硬件，才不容易犯错。而对于顺序执行，则可以采用堆栈、指令流等技术来进行优化，用以提升硬件的性能。

但是当用面向过程的方式来思考现实世界人与人之间的分工的时候，就会遇到很大的困难。比如，流程的每个节点是分散在不同的角色身上的。用面向过程的方式来思考流程的时候，不同角色的界限就没有了，他们的职责和权力就会淹没在流程中。而在现实中，人们经常会对不同角色的权责进行调整，面向过程对此往往无能为力。因此大家会发现，在角色较多且关系较为复杂的业务领域，如果完全采用面向过程的方式，经常会搞得一团糟，一旦需求有任何变化，就会使得所有人都疲于应付。这是为什么呢？

21.2 什么是面向对象

人们开始意识到，这个世界不仅仅只有过程，还有其他的特质，那就是对象。在计算机软件领域就出现了面向对象（Object Oriented）的概念，随之出现了一批面向对象的编程语言，其中最具代表性的就是 Java。

那么什么是面向对象呢？

在前面已经介绍过了，任何事物都有其生命周期，生命周期按时间推进的特性形成了面向过程。生命周期还有另外一个特性，即在事物生命周期推进的过程中，生命周期活动的主体是不变的，其生命周期活动的承受者就是该事物本身。这就是权责对等，也就是所谓的“自作自受”，有权力“作”，就要有相同的义务承受“作”的结果。比如一个人从生到死的整个生命周期过程，别人是没办法代替的，必须由这个人自己承受，权责是对等的。所有的生命周期活动都作用并积累在该事物的本身，这就是面向对象。这一特性也被称为面向对象的“内聚”。

有了面向对象，人们就能够在代码世界中，用对象来描述人类在现实生活中的分工（即生命周期的拆分），并且能够用对象来描述自然界的分工。这在编程领域，又跨了一大步。相对于面向过程的方式，面向对象的思考可以让软件工程师编写更加复杂的软件，也能够更完整地表达这个世界。这使得软件工程师的思维可以更加接近人类本身的社会生活，从而用人类的语言，在软件中表达业务和模拟业务。

21.3　生命周期和面向对象及面向过程

由前面的讨论可以看到，面向过程反映的是生命周期按时间推进的特质，而面向对象反映的则是事物生命周期活动内聚的特质。

两者不是对立的，而是各有各的特点，反映的是同一个事物的不同侧面。用一个更恰当的词语来说就是“非一非二”：两者既不是同一个，也不是两个不同的东西，而是同一个东西的两面，不可分割。

只采用面向过程的方式为何不能更有效地表述这个世界呢？这是因为过程只是其中一个特质。过程本身一定是某个主体生命周期中的一个过程，也就是某个对象的一个过程。面向对象的方式表述了事物的内聚，但是面向对象不能离开面向过程的基础，也就是顺序执行。对象的生命周期推进一定是通过一个个过程实现的。所以去看看所有的面向对象的编程语言，就会发现代码块还是顺序执行的，最终也还是交给处理器顺序处理的。因为最终执行还是要按照生命周期规律、严格按照时间顺序发生的。这也是这个世界的两个特质：首先，时间是严格按照顺序推进的，生灭是不可打破的；其次，每个事物的生命过程都要自己承受，也是不可打破的。

21.4　架构和面向对象及面向过程

一个生命周期拆分之后，会形成多个子生命周期。也就是说，一个对象拆分之后，会形成多个对象，每个对象都具备自己生命周期的所有活动，这就是对象的拆分。生命周期的架构拆分，在软件中表达为对象的拆分。比如第 20 章中的代码例子，其中，订单对象、账户对象都是用户生命周期中的活动。用户对象拆分出订单、账户对象之后，就成为用户、账户、订单三个不同的对象，形成了三个不同的生命周期，而用户是核心生命周期，账户和订单是非核心生命周期。

拆分之后的合并，则是通过面向过程的代码来串联，形成访问流程，也就是访问生命周期。最终还是完整地完成了原有用户的所有生命周期活动。但是拆分之后，不同的业务人员可以分别关注不同的生命周期，并行地推进用户的访问生命周期；不同的代码人员也可以也分别关注不同的生命周期，并行地编写代码，从而提升代码产出的效率，使得不同的代码人员之间不会互相干扰。这就是因为“内聚”而导致的“松耦合”。

因为这一拆分，代码就分为了两个部分。访问型的代码在靠近用户这一侧，因为用户的访问活动是流程性的，即用户通过过程性的访问活动推动生命周期往前发展。比如前面代码的例子，服务代码就是过程性的。而描述业务生命周期内聚的代码则是软件系统的核心，也就是业务的生命周期拆分所形成的分工，比如用户、订单、账户等。这样，生命周期活动的两个特质就都具备了：

- 服务提供用户操作的界面，并通过业务流程来组合不同的生命周期活动，即对象的方法，推进生命周期活动向前发展，并最终积累到每个对象的身上，也就是每个事物的本身。
- 用户则通过推进访问的生命周期，完成对服务的调用，以便推动自己的生命周期活动。同时用户对每个对象推进的结果都最终累积到该用户自己身上。

大部分时候，业务的变化都是流程的变化，并且都是和用户打交道的部分，也就是用户访问生命周期，所以这一部分的变动最频繁。

软件开发的变动基本集中在这里。可是不管这一部分如何变动，都不会影响到分工。因为分工并不是都由流程造成的，恰恰相反，是分工的出现才形成了流程的组织。因此，这部分的代码要尽可能地采用组合，本身不要有业务逻辑。要通过组合业务的对象来完成自身的流程，组合可以让软件工程师快速地应变。

业务对象是很稳定的，因为分工的变化并不是经常发生，其内在逻辑也是相对很稳定的，值得依赖。

一旦业务发生新的架构拆分，流程本身就要跟着改变。

这就是软件开发的根本，其本身还是对事物生命周期的反映，是对现实生活中业务架构的认识，所模拟的还是这个现实世界。事物的生命周期就像滚滚长江水，一去不复返，川流不息，谁也无法阻挡。就如易经“乾卦”所说：“天行健，君子以自强不息”。软件开发者们当然要：“软件法人”、“人法地，地法天，天法道，道法自然”。

21.5 面向对象的误区

大量讲面向对象的书籍，指出面向对象语言的三大特性，继承、封装和多态。

继承和多态，更多的是指事物的共性。只有封装和面向对象的内聚有关系。而要想真正做到面向对象，就必须要去亲身体验事物的个性，也就是事物本身独特的生命周期，才能够真正地明白对象本身。就如前文所说的红玫瑰，它独特的红，必须要亲身体会才行。古人云：“如人饮水，冷暖自知”，“夏虫不可以语冰”，说的就是这个道理。

事物有其共性，也有其自身的个性，或者说特性，最关键的就是其自身的特性。正是因为事物有其独有的特性，所以它才有一个独有的名字，才配得上一个名字，也因此才能够有一个独立的对象。这个对象当然要把这个事物的独特生命周期表达出来才符合这个事物。即便是共性本身，也就是共性这个名字所指代的事物也有其独特的个性。

比如容器和碗，容器就是碗的共性。碗有其特性，容器也有其特性，两者是不同的事物，具备不同的特性。容器虽然可以作为碗的共性，但不能代表碗。在软件编程中，经常用容器来代表碗，并称之为“多态”。这其实是一个在代码上偷懒的办法，往往会造成很多的问题。因为当人们真正需要碗的时候，还要把容器强制转义为碗，这时常常会出错。而且只有在运行期才能发现这个问题，此时，就会发现很难定位问题的原因，最后给上线发布造成很多不必要的麻烦。

继承更是有大量的其他问题，就不一一展开了。所以大家都很强调组合就是这个原因。组合实际就是面向过程，组合其他拆分出去的对象来完成自己的生命周期。

组合实际上已经包含了面向过程和面向对象的两个特质，因为组合要使用拆分的对象来完成过程的实现。

有人会问，对象应该划分为多大才合适呢？要明白，对象的大小不是对象的分割因素。对象的设置要和现实的生命周期拆分，也就是分工相匹配。难点其实是要找到现实中相匹配的分工实例，而分工实例就隐藏在所对应的业务领域分工中。

架构师要解决对象划分的问题，一定要先对现实生活中的业务有亲身的体验才行。任何人都无法通过别人的描述来理解业务的特性。很多架构师和软件工程师在设定对象的时候往往感到很困扰，问题就在这里。不亲身体验业务生命周期，就无法理解分工，也就是业务的架构拆分，也就无法识别生命周期的主体。一旦

业务分工确定之后，生命周期的主体就确定了，对象的边界也就确定了。该对象的生命周期活动就是该对象的行为，也就是方法；该对象行为的结果是该对象的状态，也就是数据。

21.6 对象和生命

既然对象是有生命周期的，那么对象就代表了一个生命。生命的特质就是，生命的出生总是来源于另外一个生命。比如妈妈生小孩，一棵植物的种子产生另一棵植物。然而，在面向对象领域，对象的创建不是对象本身，而是对象的管理者。比如一个新的员工的产生，就是招聘这个员工的直线管理领导。

同样，一个对象生命周期结束后，也无法自行处理后事，此时往往由其管理者来清理。

因此，面向对象语言的出现，使得用对象来模拟这个世界的每个独立生命周期成为了可能，让软件可以更加自然、轻松地模拟现实世界。

第 22 章　软件架构与设计模式

软件工程师做到一定程度，慢慢就会接触到设计模式。初学设计模式，会惊为天人，然后以为设计模式就是架构。但慢慢到一定阶段后就开始习以为常，不再谈设计模式，而是开始孜孜不倦地学习架构。这是一个很有意思的变化过程。那么设计模式和架构之间是怎样的关系呢？

22.1　模式以及模式的意义

首先来看看什么叫作模式（Pattern）。模式在中国古代指的是事物标准的样式。模式也作为英文 Pattern 的翻译，Pattern 在维基百科（WikiPedia）中的定义是："is a discernible regularity in the world or in a manmade design"，意思是，在自然界或者人类设计中产生的一种可识别的规律。其重要的特征是重复性和周期性。比如一个印章，可以不断地印出同一个式样，这就是一个模式。能够产生模式的一般被称为模板（Template）。因此在很多领域就会有许多带有模式的词语，比如行为模式、色彩模式等。设计模式是其中的一种，还有人提出架构模式等，这里不再一一展开。

模式是人类认识自然界和人类社会的一个重要手段。为什么这么说呢？这个世界虽然在不断地变化，但是所有变化的背后都是整齐划一的规律，那就是生灭。每个同类物种内部的不同生命的生命周期都是相似的，用现代科学的话来说，那就是它们都有类似的 DNA。树为什么被称为树，是因为这些树都有类似的特征。小到雪花、图案、声音等，通过同样的模式不断地重复自己，构建成一个多彩的世界。正如古话所说："物以类聚，人以群分"。这些类似的现象表现出来的都是模式，可以帮助人类认识世界。

现实生活中虽然有重复，但往往每个重复总有其差异的地方。要想得到模式，往往要结合抽象的思维。通过分析不同事物的共性，去掉差异，提取共同点，从

而得到类似的规律，可以更好地帮助人类认识规律。也就是说，模式大部分时候都与抽象思维有关。

22.2 什么是设计模式

前面讨论的是模式，而设计模式则是把模式的范围缩小限定在设计的范围。那么什么是设计呢？设计是英文 Design 的翻译。就中文来说，“设”，施陈也（《说文解字》），有布置、安排的意思，“计”，算也（《说文解字》），有核算、谋划、策略的意思。设计可以理解为一种有计划、有目的的活动，针对问题，通过对人或物进行合适的摆放，做出一个解决方案，并且还要考虑实现的成本。比如图案的设计，就是根据人对图案的敏感性，对点和线等基本图案进行合适的摆放和组合，形成自己独特的图案。

但是，不管是哪种设计都有以下两个共同点：

- 设计是用来解决人的问题的。
- 设计往往要借助已有的东西为基础，对它们重新进行合适的定位和组合，来达到或满足人的目标。

所以，设计都是人为的，所为的也是人。设计受限于当前已有的东西，也就是当时的自然条件和人类社会的技术能力。人们经常说设计不能是“空中楼阁”就是这意思，设计必须考虑到落地才行。当然也有些设计是天马行空，想象力丰富，但却无法落地的。这些一般都是个人爱好，不会在工程上讨论，但在艺术和思想上可能很有价值，可以引发人的思考。随着技术的发展，未来也是有可能实现的。

一旦把某个设计作为模式，那么这个设计一定是可以落地实现，而且很可能是已经实现了，并具有良好的效果。这个设计就会被当作模版，作为被解决问题的一个典型解决方案。这类设计被称为设计模式。

从设计角度来说，当一个设计方案解决了一个问题的时候，这个设计方案就相当于一个成熟的案例，可以作为一个知识传承下来。另一个人遇到类似问题的时候，就可以直接采用这个解决方案，或者稍作修改即可。所以模式对于人类来说，是一个知识传递的方式。

设计模式这个词汇最早来源于建筑架构师克里斯托弗·亚历山大（Christopher Alexander），在他 1977 年出版的一书中（《建筑模式语言：城镇，建筑，构造》，*A Pattern Language: Towns, Buildings, Construction*，1977）一书中，认为建筑的设计可以总结为 253 个设计模式，形成一种模式的语言，用这个模式的语言就可以构建几乎所有的建筑。

要理解这些模式，就需要理解建筑的目的就是为了给人提供独立的空间。当人类社会形成后，就会形成村庄、城镇、城市。这些建筑设施都是为人服务的，自然需要考虑人和自然界的特性。比如人需要阳光，所以建筑需要朝阳，在北半球就需要朝南；人身体的特性是要远离潮湿，所以房屋的地基表面要高于地面，大厅要高于地面；因为大厅要高于地面，自然就出现了台阶，用于和地面建立沟通，方便人的通行；大厅和外界之间又需要一个过渡，这就需要一个走廊；为了给屋顶排水，需要屋檐；水是人的基本生活需要，所以村庄和城镇会坐落在河流的附近；这些都是设计模式，也是人类为解决自身的需求而自发行成的设计。在古代，这些叫作“风水”，也称为“堪舆”[1]，实际上就是建筑基本设计模式的结晶。

当一个解决方案已经成为模式的时候，解决方案就变成了一种技巧、技术。再如在印染的时候，往往会采用同一个图案不断地重复，从而变成一个大的图案，这就是一种技术。所以当人们在使用一个设计模式的时候，实际上就是在使用一种已经存在的技术来解决现实的问题。

22.3　软件设计模式

受到克里其托弗·亚历山大的启发，四个软件架构师（被称为“Gang of Four，GoF”）在 1994 年合作出版了一本书，即《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*，1994），其中描述了 23 个通用的设计模式，用来指导面向对象语言代码的编写。由此在软件领域引发了一场设计模式的热潮，并扩展到了架构模式、流程模式，等等。

在这本书中，主要描述了面向对象软件开发中的三类设计模式：创建型（Creational）、结构型（Structural）、行为型（Behavioral）。要想理解这些设计模式，还是要回归到为人服务的角度，才能够比较容易地理解这些模式。同时也

1　堪舆（kānyú）：堪，天道；舆，地道。堪舆即风水，汉族传统文化之一。

需要注意，这些设计模式都是来源于现实生活，如果对现实生活观察得不够仔细，是无法理解这些模式的。

比如创建型的设计模式，实际上就是指如何生成对象。人类是如何生成东西的呢？产品的大规模集中生产，主要是在工厂。产品的生产是周而复始的，每件同类产品都是一样的，这本身就有一个模式。根据所生产的产品复杂度的不同，产生了不同的生产模式。如何更好、更有效率地生产，就成了一个问题。为了解决这个问题，就有了工厂的不同组织模式。

22.4 设计模式和架构

创建型的模式用生命周期来理解，就是把产生对象的生命周期单独拆分出来，即发生了架构分拆。结构型的模式则专注于对象的不同组合方式，而行为型则主要针对对象之间的沟通。

这些设计模式无一例外的都是把原有对象的访问生命周期拉长，增加了环节，在不修改原有对象的基础之上，添加新的能力或者获得新的自由度。也就是说，在原有对象的访问生命周期上发生了架构拆分。比如适配器（Adapter），好比亚洲人去欧美出差，电源插口不匹配必须转接一样，这就增加了一个环节。增加环节的好处在于能够在访问对象的时候，组合其他对象的服务以提供额外的功能。比如命令模式（Command）整合了 Action 和参数，策略模式（Strategy）组合了不同的策略来做切换等。

站在原有对象的角度来看，设计模式基本都是组合现有对象的能力，通过对原有对象访问生命周期进行架构拆分，形成一个新的解决方案来解决特定的问题。站在业务的角度，人们创造一个事物的过程总是先建设好事物本身，再考虑如何更好地把业务和用户连接起来。设计模式正好契合了连接用户和业务的这个需要。当业务及其架构拆分已经成型，怎样让这个业务到达用户？而用户的需求变化又比较频繁，如何尽可能地减少用户和业务各自变化的互相干扰？对于这些问题的解决，也促进了设计模式的发展。

因此，设计模式所要解决的问题和原有对象所要解决的问题并不相同。站在组合的角度，设计模式是集合现有对象的能力，针对要解决的问题，通过某种方式的组合来形成问题的解决方案。这或许就是原作者为什么要在书名中加上“可

重用”（Reusable）的原因，即尽可能利用现有对象的能力，通过某种形式的组合，形成新的对象和能力提供给外部。

为了组合原有对象形成新的能力，势必要对原有对象的能力有所取舍，那么就需要很强的抽象能力，把所需要的能力抽取出来，摒弃不需要的部分。所以理解设计模式需要很强的抽象能力。

从设计模式的分工角度来看，设计模式需要不同的对象结合起来完成一个目标。也就是说，一个设计模式是有一个很清晰的内部对象分工的，这个分工完全取自于现实生活，和现实生活相对应。这是因为设计模式所解决的问题和现实生活中所要解决的问题类似，所以设计模式内部的分工其实是在模拟现实生活的分工。并且设计模式内部的分工往往更加的抽象，是更简化的方式，而现实生活往往更加复杂。因为只有去除了具体问题的独特性，才能让设计模式更普遍地重复使用。

结合前述的代码架构可以发现，在业务领域代码的对象组织中，业务领域代码的对象分工是和现实生活的业务分工保持一致的。所以在编程时，业务领域是已经存在的。但是对于服务、黏合代码和存储这三部分，在代码中并没有对应的业务领域，也没有对应的分工来进行协作，然后和用户的请求对接，这就有很大的问题。比如，如何把用户的请求转变成对服务的调用？黏合代码如何结合业务领域对象和存储形成一个具备记忆的虚拟人？存储代码如何和存储打交道，而不影响业务领域？如果存储设备变化后对存储代码的影响是什么？如此种种的问题就会开始困扰代码编写人员。

而如何组织业务领域代码的对象来完成业务流程，同样也是有自己的业务模式的。这些模式基本都集中在软件的访问生命周期中，夹在用户和业务领域之间，这就是软件设计模式主要针对的领域。当理解了这些问题的场景后，设计模式就可以帮助到软件工程师了。比如黏合代码是为了给业务对象加上状态，这时就可以用装饰器（Decorator）；服务很多时候扮演的就是代理（Proxy）；存储基本上就是适配器，等等。设计模式主要集中在软件本身的领域，帮助提升访问代码的编写效率和质量。

由上述分析可知，这些设计模式在最开始出现的时候，是需要做架构拆分的。也就是说，软件设计模式本身就是一个架构拆分的结果，只是这个拆分被标

准化了，可以被重复使用而已。而设计模式在被使用的时候，则不需要再进行设计，直接使用即可。因此设计模式就变成了一个成熟的技巧或者技术了。

软件设计模式这部分的代码其实是有自己的业务领域的，这个领域就是软件的访问生命周期。

这个领域隐藏得比较深，并且往往和所服务的业务领域并不相关，所以人们经常会忽视它，并把它当作技术来思考。软件技术的提升往往也集中在这个地方。明白了这一点后，就知道设计模式应该用在什么地方，使用起来也会更加如鱼得水。

一旦这部分访问代码中混入了所服务业务的逻辑，就会导致访问代码部分自身的业务领域更加的模糊，访问代码部分自身的业务和所服务的业务混杂在一起，无法区分。这就是为什么软件工程师和软件架构师的时间大部分都花在访问代码这部分上，整天被技术驱赶，疲惫不堪的原因。从这里也可以看出，把所服务业务的逻辑从访问代码中剥离出来是多么的重要。只有从访问代码中剥离了所服务业务的逻辑，才有可能讨论软件访问生命周期自身的业务模型。后续软件访问生命周期部分，会展开对这个业务领域的进一步讨论。

22.5 设计模式的误区

首先从模式本身来看，是用来认识世界的。一个模式重复周期性的出现，是外在的表现，这些类似的现象是人们对共性抽象的结果。但是一些模式外在的表现一样，并不代表它们背后产生的原理也一样，解决的问题也一样。比如生产产品，妈妈生小孩，都是生产，但是两者的特性是不一样的，不能把妈妈生小孩搞成工厂模式，妈妈不是生产机器。

另外，同样一个问题表现出来的模式可能并不一样。比如，同样要解决房间进出的问题，浴室和大厅的处理就不一样。浴室空间小，采用推拉门最合适。而客厅比较大，要经常进出，则采用旋转门较好。如果浴室很小也搞旋转门，就很难办了。

如果仅仅从应用模式的角度来考虑，就会造成很多的笑话。

毕竟模式只是关注到了共性，共性只会让解决问题变得更容易、更轻松，因

为别人解决过了，有参考，但无法解决真正的问题。而真正要解决问题则是要发现问题的个性，发现了个性，共性才能发挥更大的作用。

很多人刚开始接触设计模式时，以为这个就是“银弹”，恨不得所有的地方都用模式来解决问题，反而导致代码混乱不堪，人为的造成更多问题。这就是所谓的“过度设计”。

设计模式是一种技术，是用来传承设计思考的。通过对设计模式所解决的问题的分析，可以看到作者是如何思考的，作者是如何应用现实生活经验，去拆分对象、组合不同对象的，为何是映射这样的生活场景，而不是另外一种？这样思考下去，就会慢慢接近前辈的视角，才能够真正的受益。

要真正用好设计模式，就要和使用技术一样，不但要理解设计模式所能解决的问题，还要深刻地理解自己要解决的问题。如果两个问题是一致的，那么设计模式才可能派上用场。但这只是发现了共性，仅仅有共性是不够的。接下来还要考虑自己所要解决问题的特点和个性，设计模式的共性会不会给要解决的问题的个性带来阻碍或者干扰？还是不影响或者加强？如果是阻碍，则不要采用。仅仅从功利的角度生搬硬套来解决表面的问题，往往会造成更大的问题，这是大部分架构师和软件工程师所犯的错误。

还有一个误区是架构师和软件工程师普遍存在的一个省力思维，总想用一个不变的办法来解决所有的问题，什么代码都考虑重用。设计模式强调的也是重用，恰好满足了这一需要，因此存在滥用的状况。但是重用并不总是带来好处，比如不同角色重用同一个服务（Service），就会导致两个角色不必要的互相干扰，反而增加了新的问题。重用并不是软件开发的目标，软件开发的真正目标是模拟业务，并提供用户的访问。

对于业务对象来说，业务对象表达的是业务的核心分工，是不同用户所访问的目标，这部分自然是要被重用的。因为现实生活中，业务的分工就是这样的，执行业务的时候按照业务的访问流程组合不同的对象即可。但是对于服务、黏合代码和存储而言，它们都有自己独特的业务问题，即它们处理访问通道的，目的并不是给大家共享，也不是访问的目标。而处理好通道问题，则是要按照不同角色的用户来进行分析，提供不同的通道，让他们之间的访问互不影响，才能够服务好不同的要求。访问通道是不能重用的。

很多架构师或软件工程师使用了很多设计模式，就是为了让访问代码部分能够做到重用。究其根本原因，是因为这部分代码里面混入了业务逻辑，并且深深地认为这样做是对的，因为只能这么干，并且所有人都这么干。结果当业务需求快速变化的时候，设计模式因为无法规避不同用户修改需求带来的相互影响，反而导致访问部分的代码修改更加的困难。因为设计模式让访问代码变得更复杂了，最终事与愿违。

总而言之，设计模式是人为抽象的，具有共性，既可以作为技术来使用，也可以认为等同于技术。软件中的业务技术或模式应该和现实生活中业务技术或模式保持一致。软件设计模式同样来源于现实生活，也有自己对应的现实生活的业务，即用户访问生命周期。访问生命周期的特性留待后续进一步讨论。

第 23 章　软件架构和软件框架

要把业务呈现给用户，并让用户能够独立访问，形成软件，会遇到两大问题：

- 一是如何把业务用软件表达出来；
- 二是软件如何被放入计算机，提供给用户访问。

第二个问题是所有的软件都会遇到的。为什么要提供给用户访问，是因为计算机本身是没有意志的。计算机和软件结合能模拟人的行为、模拟人的记忆，但是无法模拟人的意志。所以计算机软件内部所模拟的生命周期变化，都是要由外部被模拟的人通过个人意志来推动。而在推动的过程中，通常会有下面这些问题：

- 提供给用户什么样的访问界面？
- 用户访问时，如何和用户进行交互？
- 用什么方式组织业务逻辑并对外提供访问？
- 业务逻辑的状态如何从存储（Repository）中存取？
- 软件以什么方式在计算机中运行起来？

针对这些问题，就会有一些共同的业务模型，形成一些设计解决方案。这些方案中又会出现一些典型的设计和解决方案，从而形成设计模式。

23.1　访问类框架

针对用户访问的问题，设计模式的典型作用就是把用户的访问和最终的存储连接起来，形成访问通道，为用户访问业务逻辑提供便利。因此模式都有一些典型的特点：对用户访问输入端会提供访问的方式，并且这些访问点可以横向扩张；对通道下一节点访问的输出端也会横向扩张，而其本身则是实现这个设计模式背后的业务。

针对上述这些设计模式实现的时候就会发现，经常进行修改的部分，往往是

输入输出部分的扩张，比如添加功能或者新的访问点。因此往往会把经常变化的这部分独立出来，使得扩张简单化，甚至是通过配置就可以实现。其核心部分则是实现设计模式本身，并为输入输出扩张的简化提供支持，同时为输入输出提供数据传输和转换的通道。慢慢大家就会发现，核心部分是非常稳定的，并且可以用在不同的软件中。有些人就把这些稳定的部分开源出来分享给社区，成为一个通用的代码解决方案。这类成熟的代码解决方案被称为框架。

比如大家常说的 MVC 框架（Model、View 和 Controller），就是基于 MVC 设计模式的一个成熟的解决方案。其核心是为了方便用户对内部业务逻辑进行访问，提供给用户视图来做交互，并分别对视图和模型的变化提供支持，通过简单的声明或者配置就可以很容易地更改。而对控制器（Controller）则进行了封装，作为其自身的核心业务，对用户不可见，即变成透明的。这样用户在开发用户访问界面时，就不需要自行处理所有的环节，只需专注于视图和模型本身的对应部分即可。因此，用户的访问生命周期就被 MVC 做了切分，即把控制器拆分了出来，也就是说发生了架构拆分。

很多人把 MVC 中的模型（Model）理解为业务模型，这是有问题的。模型更多的是指对视图（View）的数据支持，一般用 DTO（Data Transfer Object）来表达。而业务模型关注的是业务生命周期及其行为，业务模型的内部数据只是这些行为的结果。MVC 中的模型和业务模型是两个不同的概念，不可混用。但是两者可以通过数据转换结合起来连接沟通使用，需要类似于适配器（Adapter）的模式来解决这个问题。

再如典型的 ORM（Object-Relational Mapping）框架 Hibernate。在面向对象软件开发时，会遇到一个问题：数据库存储是关系型的，和面向对象并不一致，如何保存对象的状态？这就需要把对象的状态转换成数据库的存储，或者把数据库的存储恢复为对象的状态。这两者的适配自然就需要有一个转换，需要类似于适配器的模式来解决问题。同时，存储的变化和输入的变化也需要能够支撑，根据不同的请求来匹配映射（Mapping）存储。

仔细分析还会发现，所有的通道型的框架，背后都有控制器的特质，以方便前后端接入点的扩展；也会有数据转换的要求，以方便前后端数据的传递。只是在不同的场景下，侧重点不同而已。这是由通道型业务本身的特性所决定的，通

道的目的就是连接前后端并传递信息。

23.2　业务类框架

对于有些企业，为了给整个行业提供解决方案，既要考虑整个行业的业务模式，还需要适应行业内不同企业的区别，因而往往会形成一个行业的框架。比如 CRM（Customer Relationship Management），很多企业都是需要和客户打交道的，但是不同的企业又都有自己的业务特殊性。CRM 框架就是把和用户打交道的基本规则封装起来，然后留给各企业按照自己企业的特点做一定变化的空间，最终所形成的就是一个行业的框架。

行业类的框架一般都是由行业解决方案提供商来整体提供。

23.3　什么是框架

什么是框架呢？框架基本上都是根据业务模型，或者设计模式等，把模型中稳定的部分进行封装，形成一个大的边界，但是具体内容仍留有余地。由于业务模型或者设计模式也是架构拆分的结果，因此框架同时也属于业务架构的一个具体实现。框架对业务模型中变化的部分，一般都会提供很容易的扩展方式，使得框架使用者可以根据自己业务的特殊性对框架进行扩展，而不需要全部从头开始编码。因此软件工程师不再需要从零开始编写代码，他们可以站在巨人的肩膀上工作，极大地减轻了软件工程师的工作量，提升了开发的效率。

形象点地比喻框架，就好比建筑里面一扇窗的框架，窗户的整体已经具备，整个边界已经清晰，但是窗户的内容则可以由用户根据自己的需要自行进行改造，比如是使用玻璃还是木板，推拉还是转轴等。

再举一个软件中的例子。在没有 Struts 之类的 MVC 框架的时候，Java 的 Web 开发效率是非常低的。每个软件工程师都需要自行扩展 Servlet，需要关注用户访问链条上的每个节点。Struts 的出现让用户的访问通过简单地配置即可接收到用户的请求，实现对用户请求的处理。这样软件工程师就可以专注于业务模型的思考，同时业务逻辑也就从 Servlet 中根除了。Struts 形成了一个边界，把访问逻辑和业务逻辑分离，形成了对访问链条的架构拆分，同时其内部也形成了架构拆分，通过控制器把用户的访问和业务服务连接了起来。

23.4 框架的特点

框架往往都是无法单独运行的。软件框架基本都是一个留有扩展余地，为其他代码所引用的代码或类库。比如一扇窗的框架，虽然有一个窗的样子，但是无法直接使用，还需要配上能挡风的设施。如果是可以单独运行的，一般会被称为服务。如厂商提供的是一扇完整功能的窗，那么就属于窗的服务，基本没有定制空间，拿来用即可。

框架往往是为了方便本地定制的，在本地进行改造，和自己的软件结合在一起。再比如一个内容管理框架，可以帮助应用管理静态内容，提供文件管理、版本管理等功能。软件根据自己的需要整合内容管理框架，可以让自己的软件具备内容管理功能，只需少量地开发对接即可，不需要从头开发。如果把内容框架单独运行，则会变成一个内容服务或者 CDN（Content Delivery Network）的服务，甚至是一个内容服务的云。

在软件行业，框架和服务的另一个区别在于：软件引用框架是本地引用的方式，而服务是用来远程调用的。

框架的背后总是有一个模式存在，背后的原因是因为有一个大家普遍都会遇到的共同问题。针对这个问题往往都有一些通用的设计模式，有一个和现实社会对应的架构拆分，形成了一个自己的业务领域，并普遍为大家所接受。如 MVC 框架、Hibernate 框架等。

框架的目的是方便让其他软件在内部解决问题，只要引用了该框架，就自动获得了这个框架所提供的能力。而要想应用好这个框架，首先需要理解好这个框架所能解决的问题，然后才能理解这个框架本身的业务模型和架构的拆分方式，最终才能把其他人的工作变成自己软件的一部分，此时才能“站在巨人的肩膀上”工作。

总之，框架的背后都有一个模式，都有一个架构的拆分。框架基本都是某个架构的实现，但是留有扩展的余地。比如 MVC 就是一种架构的模式，简称 MVC 模式，是对访问生命周期某个阶段的架构拆分结果，是这个架构分拆结果的模式化和标准化，而 MVC 框架则是对 MVC 模式的实现。因此，MVC 模式就成为了一种技术，而 MVC 框架则是这个技术的具体化、工具化，它成为了一个工具。

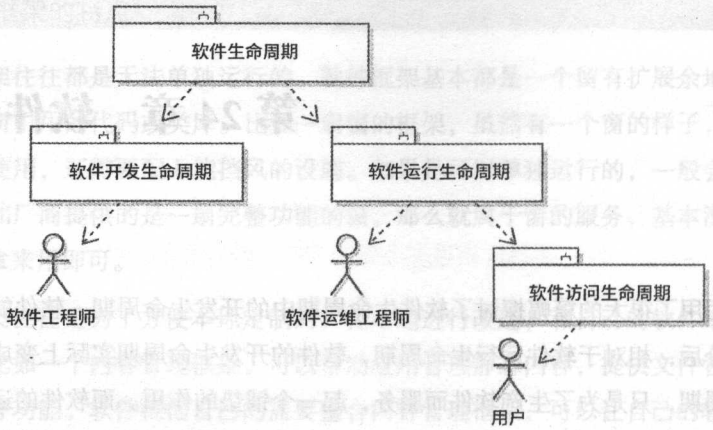
第 24 章　软件运维

前面用了很大的篇幅探讨了软件生命周期中的开发生命周期。软件的生命周期在拆分后，相对于软件运行生命周期，软件的开发生命周期实际上变成了非核心生命周期，只是为了生产软件而服务，起一个铺垫的作用。而软件的运行生命周期才是真正的软件核心生命周期。这就是软件的开发可以外包、软件的代码可以开源的原因。但是很少有企业愿意把自己的运维暴露给外部，并且往往对自己的软件运维部分严防死守，因为，运维才是软件真正的核心竞争力。

24.1　软件运行生命周期

软件的运行生命周期从软件启动开始，不断地运行，积累数据，直到软件被终止运行。在软件的整个生命周期中，会经历多次的软件运行生命周期。而软件的运行生命周期所要达到的主要目的，则是为了能够提供给用户持续不断的访问，因此软件运行生命周期的核心是软件访问生命周期。

在软件的运行生命周期中，会经历多次的软件访问生命周期。通过用户的访问不断地积累用户的访问数据，因此访问生命周期是核心生命周期。软件运行生命周期是为软件访问生命周期服务的。因此形成了如下图所示的软件生命周期分拆树：实线代表树的主干，也就是核心生命周期，树的遍历顺序从左至右。



其中，软件开发生命周期主要由软件工程师来推动，软件运行生命周期主要由软件运维工程师来推动，软件访问生命周期主要由用户来推动。

软件的访问生命周期，从软件接收到用户的访问请求开始，执行用户的请求，到返回给用户请求结果为止。

24.2 什么是软件运维

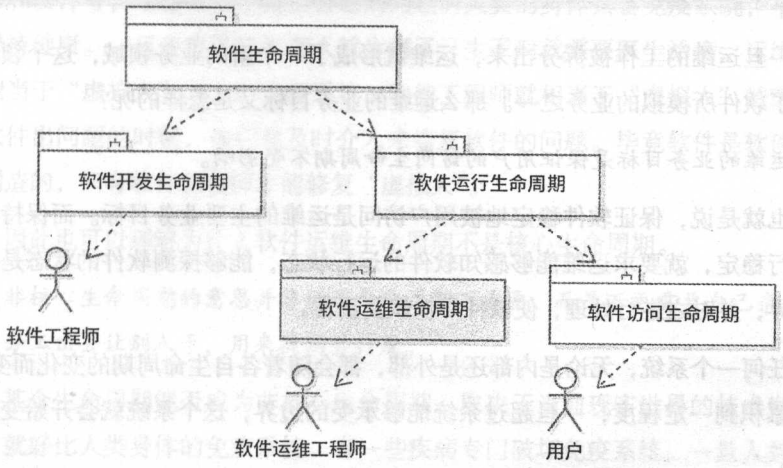
什么是软件运维呢？软件运维其实就是指软件的运行和维护。

在软件生命周期没有拆分之前，是没有分工的，软件工程师干所有的事情。在软件生命周期拆分之后，软件工程师专注于软件的开发，从而形成了如上图所示的拆分树。为了服务好软件访问生命周期，软件工程师所做的事情就发生了分工：软件的启动、停止、版本更新等运行生命周期活动，需要由独立的工程师来进行维护，这就是软件运维工程师。

为了维护好软件运行生命周期，运维工程师需要安置相应的传感器来搜集软件生命活动的健康状况。比如软件是否存活？用户的访问生命周期是否正常？等等。这些运维监控软件需要由软件工程师进行开发，它是一个独立的业务领域。运维工程师角色独立出来后，运维监控的生命周期就从软件的运行生命周期中拆分了出来。因此，软件运行生命周期的推动分为两个角色：一个是运维工程师，另一个是用户。

软件运行生命周期也可拆分成：软件运维生命周期和软件访问生命周期。如

下图所示。其中，软件访问生命周期是核心生命周期。运维生命周期从软件开始部署为始，到软件结束服务为止。



由此也可以看出，运维工程师本质上也是一个软件工程师，只不过专注的领域是计算机软件本身的运营和维护，与软件工程师所针对的业务领域不同罢了。一旦运维软件变得很成熟，软件工程师就可以代替软件运维工程师的工作，这个角色也会逐渐地回归到业务的软件工程师，发生树的节点的合并。这就是所谓的自动化，和业务软件的自动化没有太多的区别。

因为软件本身也是软件的业务。

从最开始只有软件工程师，逐渐分工成为多个角色共同协作，再慢慢地又回归到只有软件工程师，分分合合，又是一个周期，但是生产力变得大不相同。世界不就是由这样一个一个的周期往前推动进步的吗？

既然运维工程师可以用软件来替代，那么软件工程师本身能够用软件来替代吗？这个基本上是不可能的。软件只是人类解决自身问题的其中一个工具而已，可以用来模拟人，但不是真正的人，因为软件不具备人的意志。人类的问题还是需要人类自己思考才能够解决。即便是人工智能，它的来源还是人类本身的意志，是人类利用软件模仿自己的产物，还是需要软件工程师来实现的。C 编译器就是一个典型的例子，虽然 C 编译器也是用 C 编写的，也在不断地改进，但是程序没有自己的意志，编译器改进的推动还是要人来执行。即便是人类自身，一个人理解

另一个人的想法都有极大的困难，更何况是让机器理解人的想法呢？

24.3 运维的业务模型

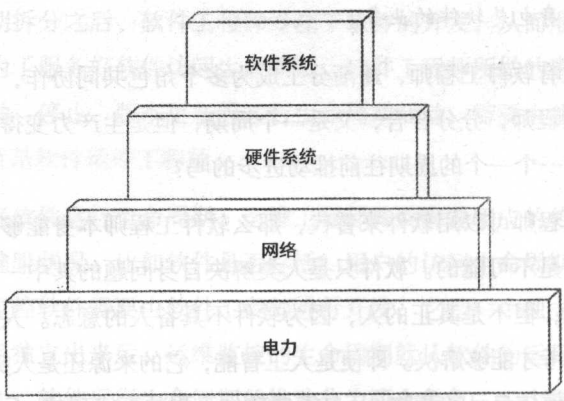
一旦运维的工作被拆分出来，运维就形成了一个新的业务领域，这个领域也变成了软件所模拟的业务之一。那么运维的业务目标又是怎样的呢？

运维的业务目标是保证用户的访问生命周期不受影响。

也就是说，保证软件稳定地被用户访问是运维的主要业务目标。而保持软件的运行稳定，就要求运维能够感知软件的运行状态，能够探测软件的状态是否超出边界，并对之进行处理，使软件尽快恢复正常。

任何一个系统，无论是内部还是外部，都会随着各自生命周期的变化而变化。变化累积到一定程度，一旦超过系统能够承受的边界，这个系统就会开始变得不稳定。

对于一个软件系统来说，软件运行在硬件上，硬件又依赖于电力和网络。软件、硬件、电力和网络任何一个变化，都会导致软件不稳定，影响软件的稳定运行。同时，软件是提供给用户使用的，用户访问量的变化也会导致软件的不稳定。如下图所示。如何控制这些变化，以及监控这些变化就成了运维的主要业务，必须先要知道变化才能够对变化采取行之有效的行动。



软件运维生命周期的业务和现实生活其实也有一个对应。一个软件就相当于一个虚拟的人，保持这个“虚拟人”生命的健康，就是软件运维生命周期的业务。

要使软件持续不断的运行，软件运行生命周期就不能停止。而人吃五谷杂粮，怎能不生病？天气变化也是容易导致人生病的原因之一。“虚拟人”则需要依赖电力、网络和硬件等，这些设施本身也会遇到问题。人类的身体具备免疫系统，平时可以保持健康。一旦免疫系统失衡人就生病了，生了病就需要医生治病。运维系统就相当于“虚拟人”身上的免疫系统，软件工程师就相当于“虚拟人”的医生。当软件出问题的时候，他们就及时介入来修复软件的问题。毕竟软件是软件工程师创造的，只有软件工程师才能修复“虚拟人”。

因此也可以理解为什么软件运维生命周期不是核心生命周期。

非核心生命周期的意思并非说该生命周期不重要，而是不再需要自己亲自干，可以分工出去让别人干，用来提高并行度。

某个生命周期能否成为非核心生命周期，取决于当前现实世界的技术发展水平。就好比人类身体的免疫系统，有一些疾病专门破坏免疫系统。一旦人类的免疫系统被破坏，人类还可以依赖于某些药物来形成抵抗力，或修复自身的抵抗力，用来继续维持生命，这些药物就是新的技术。没有新技术之前免疫系统就是人类核心生命周期，这时是分拆不出来的，被破坏了就只能等死了。新技术出现后，免疫系统就不再是人类的核心生命周期，人类的生命周期形成了新的拆分，而社会分工中就增加了药物研发生产企业这个角色。很多软件企业用第三方的软件或者第三方的云服务来实现自己软件的运维，这是一样的道理，是一个新的生命周期分拆。第三方的软件或云服务没出现时，软件运维是核心生命周期，不可分拆，必须自己来实现。第三方的软件和云服务出现之后，运维的生命周期成为非核心生命周期被分拆出去了，软件行业出现了新的分工，形成了运维软件和云服务的企业。运维生命周期分拆出去之后，第三方的软件或云服务和业务软件在运行时结合在一起，执行顺序还是和没有拆分的时候一摸一样，运维业务本身是保持不变的，重要性也没有变低。

24.4　控制变化

运维的生命周期是从软件部署开始的。为何是从部署开始呢？就像人一样，人要活的健康，就不能乱吃东西，因此人类有一个安全的食品列表。并且人要尽量待在安全的地方，防止自己的生命受到伤害，古人云：“君子不立危墙之下”。

为了要保证软件运行的健康状态，软件也是一样的。软件变更后的每次部署，就相当于“虚拟人”吃了点不同的东西，会不会导致重大疾病？某些环境发生了变动，是否会让软件处于“危墙之下”？这些是运维最关心的事情。因为任何对软件的改变，都是风险，都是需要运维关注的。

因此，每次变化的风险控制是最重要的任务。

隔离

要做好变化的控制，首先要做的是：在软件周围设立隔离区，避免软件出现在不安全的地方。通过隔离区可以控制所有对软件的改变，对软件所依赖的硬件、网络和电力也是同样的，只有设立隔离区才能保证软件的安全。

考古的人对此感受最深。比如埋藏在地下几千年的文物，此时已经和大地融为一体，很稳定了。一旦暴露在空气中，这么巨大的变化对于文物是致命的打击，因此很多文物刚出土就毁了。为了保存文物，必须在文物周围形成一个隔离区，确保文物所处的环境和地下类似，人们才能够在博物馆观赏前人的灿烂文化。同时也可以看到，越脆弱的文物越是保护严密，哪怕一点点的变化也是致命的。软件也是一样，越脆弱的软件越是需要更严格地控制软件的生存环境，以便控制环境变化对软件的冲击。

而为了要控制变化，隔离环境是第一件要做的事情。

有人会问，隔离软件还好理解，为何要隔离网络和电力呢？很多企业确实是不隔离的，因此经常会有不小心拔掉网线，或者下班关掉办公室电源导致软件系统停止的事件。如果想让软件 7*24 小时（计算机和软件行业术语，指每周 7 天，每天 24 小时持续不断）运作，整体隔离是必须要做的事情。

如何隔离

如何做好隔离呢？首先要拆分软件的运行环境。根据软件的生命周期，软件开发生命周期主要是日常办公沟通的工作，软件运行生命周期是服务于用户的，因此首先要区分出的是办公环境和生产环境（Production）。办公环境是企业内部人员使用的，包括软件工程师编写代码也是在办公环境。生产环境是给用户访问的，是企业提供用户的服务。也就是说：

环境隔离导致了架构的拆分，企业的环境隔离形成了一个架构分拆。

要避免办公环境对生产的影响，就必须划出独立的机房给生产环境使用。这个独立的机房，要有独立的电源、独立的网络、独立的空调、独立的排风、独立的门等。从其他环境来访问生产环境都必须是受控的，包括网络上从办公环境来的访问，从外部用户来的访问，以及物理上通过机房大门进行的访问。

生产环境建立之后，就相当于建立了一个软件所依存的世界，软件就是在这个世界中生存的虚拟人，随时随地受这个世界的影响，依赖于这个世界。多个软件并存在这个生产环境中，形成了一个软件的社会。

控制变更

什么是变更（Change）？生产环境一旦建立，那么生产环境就有了自己的生命周期，其内部就会开始持续不断地发生生命周期变化，这些生命周期变化就是变更。无论是网络上还是物理上，对生产环境上的访问都是变更。包括机房内机器、设备本身的老化，这些都是变更。也就是说，生产环境自身生命周期的推动、变化也都是变更。

所有生产环境的变更可以分为两类：一种是被动发生的变更，一种是企业内部主动发起的变更。

哪些是属于被动发生的变更呢？比如机器的主板、硬盘的损坏，用户访问量的突然猛增，这些都是被动发生的变更，往往是无法预测的。要控制这一类的变更对软件的影响，运维就必须容忍这些变更的发生。而要容忍这些问题，就必须先让自己具备更大的能力。比如针对硬件、设备本身的变更，要购买稳定的产品，并且通过定期淘汰硬件、设备来避免问题的发生；通过增加设备进行并联来达到冗余，即使有问题发生，还保留通路可达软件。即便如此，还是会有一定的概率出现某些变更导致软件系统的运行受到影响。针对这类的情况，唯一的办法只能是快速发现问题并解决问题，减少问题对用户的影响时长。这点会在监控部分讨论。

企业内部主动发起的变更往往是导致软件系统出现问题的主要原因。主动变更基本分为以下几类：

- 软件的变更
- 硬件的变更

- 网络的变更
- 电力的变更

根据对一些互联网企业所做的统计，主动变更所导致的软件系统问题大约占所有线上问题的 2/3 以上。其中软件本身的变更所造成的问题大约占所有线上问题的一半以上，占主动变更中的绝大部分。

要控制这些主动变更，就必须确保这些变更在指定的地点和指定的时间发生，并要让所有的运维人员都知情，这也是为什么要隔离出生产环境的意义所在。有些企业的生产环境会出现很多莫名其妙的问题，经常花了很久才定位到原来是某人做了一个很小很隐蔽的变更，并以为对其他人没有影响。未隔离的生产环境，任何人都可以在任何时候做变更，令人防不胜防。

隔离生产环境之后，所有其他的访问通道都应该用“墙”来封死，不允许直接访问。从逻辑上来看，所有的变更只能通过两个入口来进行：一个是虚拟的入口，通过网络来进行访问，传递要变更的数据，并通过命令实施变更；另一个是实体的物理入口，也就是机房的大门，机器硬件、网络、电力的物理设施变更都从这个入口访问，需要人进入操作，通过人与人的沟通实施变更。这两个入口的进入必须要得到运维的许可，任何操作都必须得到运维的允许，确保运维的知情权。不过，当机器自动化到了一定程度，全部都可以收为一个虚拟的入口，通过网络来远程操控。

24.5 监控变更

通过环境的隔离，可以收口变更的通道，让运维掌控所有的变更。但是有些变更仍然会导致线上的问题。即使没有主动变更，内部硬件或设备还是会老化，用户的访问激增等被动变更，也还是会导致线上的问题。

生产环境的线上问题无法避免，也无法完全消灭，因此必须要拥抱变更才行。而拥抱变更的底气则来源于运维对系统的自信，以及对系统运行状态的了解程度。既然线上问题无法完全避免，唯一的办法只能是减少问题产生后带来的损失。减少损失的办法，要么提前预防问题，要么缩短问题所影响的时间，两者都要求运维要具备快速发现并定位问题的能力。而要具备这个能力，就必须要时刻的感知系统的运行状态，这就是监控（Monitoring）。

什么是监控呢？如果要在现实生活中感受监控，可以到医院的重症监护室（ICU，Intensive Care Unit）去看看。

监控的目的实际上就是把系统内不同生命周期的当前运行状态，通过探测器传输出来，展示到可视或可感知的设备上，来供人查看。

监控非常重要的指标就是实时度。

去医院看病的人，大都检验过血液或者排泄物。医生通过身体某部分生化指标的检查结果，来判断患者身体内部运行的状况。如检查血液，一般需要一到两个小时才能够知道结果，这个实时度是比较低的。也就意味着系统万一有问题，几个小时之后才能够发现，这是不利于快速响应问题的。而 ICU 的特点就是能够比较实时的监控身体的运行状态，实时地展示各个主要器官的生命周期变化的状态和历史。为什么重症病人要转入 ICU，因为病人的身体太脆弱，无法经受小时级别的变化对身体的冲击，必须及时对变化做出响应才行。

人类的身体无时无刻不在进行着生命周期的变化。人类也是这个大自然的一部分，人类身体的生命周期的状态变化也与大自然的其他变化一样，就如“雁过寒潭”，不留下一点痕迹。为了突破大自然对人类的限制，增加身体运行状态信息的访问，人类必须要用相应的设备把人体内的数据实时地传输出来，存储在外部并展示。这就是对人体访问生命周期的架构分拆。

这类监控为了突破人体的限制，需要在身上增加很多特制的传感器。受限于技术的发展，这类传感器会限制人的活动范围，不方便携带。医院的 ICU 相当于个人独占并持续的体检，非常耗费资源且昂贵。人们也只能定时去体检来监控自己身体的运行状况，也就是健康状况。如果传感器的技术提升到无侵入，并能随身方便携带的水平，人体的数据就能够随时监控，为提前发现疾病或者紧急救护带来帮助。这方面未来应该会有很大的突破，智能手环等设备的兴起就是一个例子。

计算机和软件的内部状态提取相对于人体就简单多了，毕竟计算机是人类自己造出来的。提取实时数据这一点在硬件设备上不会有太多的问题，因为大部分设备本身就提供了自身状态提取的入口。对于软件系统，在开发的时候就需要考虑到软件本身的内部生命周期的拆分，对不同的生命周期要预留当前运行状态的获取入口。这样监控系统就可以通过访问这些软、硬件设备的相应入口，来获取不同生命周期的当前运行数据，实时地在监控设备上虚拟化展示出来。这和医院

的 ICU 并没有太大的差别，背后的业务原理也是一样的，只是软件做到这一点更容易，因为相当于有了一个家庭 ICU。

不留恋过去，是大自然能够得以不断持续向前发展的前提。

古代印度是不重视历史的，很多人会很奇怪甚至笑话印度的这种行为，认为印度很落后。即使把历史记下来，事物的消逝就能阻挡吗？从这一点看，印度恰恰是非常先进的，符合自然的规律。人类应该向大自然学习。但是人类又不肯放过已发生的事情，总想把消逝的东西保留下来，以期能够更长时间地拥有并获益。这个动力推动了人类社会的物质发展，创造出了大量新技术来突破大自然的限制，梦想着创造出更好的世界。可是真的会比大自然创造出的世界更美好吗？

24.6 预警变更

监控的主要目的是实时收集数据并给人展示出来。但是一个人同时所看的信息有限，不可能全天候感知监控的数据，当面对大量的监控数据和图表时，人们会很抓狂，过量的数据和没有数据是一样的效果。哪些监控数据是有问题的？哪些监控数据是正常的？要理清楚这些问题，就需要人们去理解数据。而理解数据的前提，就是理解不同生命周期主体本身的业务。

比如要监控一台计算机，就需要理解一台计算机的生命周期。一台计算机从加电服役开始，到关电退役为终，它的生命周期变化的活动情况如何，也就是它本身的业务是如何运作的？什么样的指标说明这台机器是正常的？在什么范围是不正常的？要理解这一点，就需要理解机器的组成结构，比如要知道计算机的体系结构，如冯诺伊曼架构，知道 CPU、内存、硬盘等数据和指标。所以，学习计算机的专业课还是非常重要的，在这里就派上了用场。

监控一个软件，就要知道这个软件本身的生命周期是怎样的？它对应的业务生命周期是怎样的，如何运作的？什么样的指标说明业务是正常运作的？什么样的指标说明软件本身是正常运作的？要理解这些内容，需要先理解软件，理解软件所实现的业务，才能够知道软件的访问生命周期是否正常，软件内部的业务生命周期运作是否正常。理解了软件和业务本身的含义，才能得到软件的数据和指标。

理解了这些数据和指标，监控人员就可以对超出正常范围的数据进行报警，马上采取行动，尽可能在问题造成实际损害前就把问题解决掉。这就是运维的预警。

所以预警的内容往往会分为两部分：

一部分是软件本身业务的预警，主要包括软件、硬件、网络和电力等设备。这一部分的业务人员主要是运维工程师、软件工程师和架构师等 IT 从业人员。

另一部分是软件所实现业务的预警，比如实现用户生命周期管理的软件，要对用户的注册量进行预警，实现订单生命周期管理的软件，要对订单量进行预警。这一部分的业务人员主要是软件所服务的行业从业人员，比如电商、制造业等。

生成预警的主要困难在于对业务生命周期的理解。一旦理解了业务生命周期就会发现，业务的数据会成周期性的变化，而周期性的变化就意味着可以用历史生命周期的数据作为报警的基线。用户群体的购买行为往往呈周期性的变化，有时间上的周期性。如每周周一至周五是类似的，每周周末也是类似的；每天从早到晚的购买行为也呈周期性的变化，上班时和下班后都不同。这类规律都和用户的作息生命周期规律有关。还有其他业务相关的周期性，如历史上产品本身的营销对用户访问影响情况等。利用生命周期的周期性变化规律，就可以发现不正常的变化，这是实现预警的理论基础。

因此要做出预警，就需要对监控的数据做实时地分析。但是靠人来做这个事情，效率是非常低的。通过把不同生命周期运行的正常数据范围确定出来，就可以通过软件来模拟监控预警人员的工作，把生成预警的工作自动化，把人工释放出来。因为生成预警主要是通过判断当前系统的状态是否超过预期的范围来确定，因此生成预警往往需要一个强大的规则系统，对监控数据流做实时的分析。所以：

监控是生成预警的数据基础。对业务生命周期的理解则是生成预警的规则基础。

现代的运维预警基本都是通过软件来实现的，也就是说预警本身也是软件的模拟对象，也是需要软件工程师进行开发的，也有其自身的架构。同时：

预警软件本身也是需要监控和预警的，也是预警的业务。

预警发出来之后，预警本身的生命周期需要进行管理。每个预警本身的生命周期从预警的生成开始，到预警的撤销为止。该预警是否得到了及时的处理？是否误报？预警的级别是否需要提升？对预警的处理反过来会影响规则的制定。所以预警规则负责生成预警，预警本身的生命周期是预警系统的核心。因此可以看

出，预警业务的架构发生了拆分，拆分为预警规则生命周期和预警生命周期两部分。其中预警生命周期是核心生命周期，这是预警系统的目的。

对于运维来说，预警软件是整个运维工作的核心。衡量预警软件的指标就是预警的质量。如果被监控的软件有问题时预警没发现，就说明是漏报。如果被监控的软件没问题时预警发出，就说明是误报。运维的工作和预警系统息息相关，只有预警发出来才需要运维的介入，以使预警消失为运维工作的导向。预警消失就意味着系统恢复了稳定。判断一个运维体系是否成熟就看预警系统的建设。

24.7 主导变更

监控和预警是眼睛

很多软件工程师不敢做变更，或者运维不敢让软件工程师做变更，主要的原因就是没有自信，没有自信的原因在于没有监控和预警的反馈。和软件工程师必须要做单元测试一样，单元测试为软件工程师建立了一个正向反馈环。

在生产系统做变更，也需要有一个正向反馈环，这个正向反馈环核心环节就是监控和预警。

做任何一个变更，运维都需要知道变更对系统的影响如何，如果监控到的数据和预料一样，其他的系统也没有产生预警，那么这个变更就是正常的。如果该变更有问题，可以通过监控和预警发现，快速回退变更，消除变更带来的影响。这样运维才有能力去主导变更，对生产环境的主动变更执行也就变得比较自信。

对于被动的变更，有了监控和预警，也可以快速地发现，缩短对被动变更的响应时间，快速地恢复。

现实生活中，医生同样依赖于监控和预警。比如在不确定疾病的情况下，让患者服用小剂量的药，通过数据和患者的反应来帮助确定疾病的原因。没有监控和预警，就好比人瞎了眼睛，处处碰壁，很难找到一条正确的路。有了监控和预警，人们就可以试错，运维也能看清楚错在哪里，帮助快速地纠正错误。允许人们犯错，人们才可以拥抱错误，通过犯错不断地提升自己。

要主导变更，首要任务就是要跟踪变更。要跟踪变更就需要理解变更的生命周期。变更从创建开始，到实施结束为一个生命周期。一旦变更发生回退，其实

产生的是一个新的变更生命周期。通过跟踪变更，就可以知道变更发生的时间顺序，帮助快速地定位问题。通过对一段时间内所发生变更的规律进行分析，还可以帮助优化变更，减少不必要的变更，降低变更的频度。变更本身也是一个独立业务，也是需要监控和预警的。比如对一段时间内变更过多进行报警，对同时并发的变更数过多进行报警。

发布

变更在生产环境的执行一般称之为发布，对变更进行管理的系统叫做发布系统。

软、硬件设备的发布都需要依赖发布系统，软、硬件设备可以使用同一个发布系统。

主导变更的策略主要就是让变更逐步地发生，一般被称作“灰度发布”，其核心思想就是尽可能减少变更所影响的范围。比如修改操作系统的某个参数，可以先在某台机器上先应用，如果监控和预警都正常，再按照百分比逐渐地发布到其他机器上。同样对于软件的发布也是一样，先发布一台机器，通过监控和预警的反馈再决定是否继续。

一般第一次发布，就是该变更第一次在生产环境的应用，此时风险最大。因此要确保被影响的机器越少越好，把风险降到最低。首次发布时一般都选择可发布的最小单位，也就是一台机器。对于特别重要的软件，往往会在发布 50%之后，经历一次业务高峰后再发布剩余的机器。因为即使该变更有问题，导致 50%的机器在高峰期的压力下无法正常工作，还有剩下 50%的机器能够支撑业务运作。这就意味着在建立集群的时候，必须要留有 50%的冗余容量来做缓冲。

软件要做到灰度发布，还要要求软件本身能够支持两个不同版本的代码同时运行在线上，并且保证是对用户透明的，不会影响到用户的访问。这是软件在开发阶段的时候就需要考虑的问题，主要需要考虑的问题是数据的兼容性。也就是说，新功能产生的数据要仍然能够被旧功能读取，新功能要能读取已有的旧数据。这也是必然的，因为存储的变更一定要比代码先发布上线。

为了要能够回退变更，对数据库的操作要非常谨慎，必须确保数据库的变更也是可以通过回退恢复的。这意味着数据库的变更不能修改原有的旧数据，只能

增加新的数据。这部分是大多数人都忽略的，往往导致软件不得不停机维护，导致软件发布时软件不可用，或者只能够在半夜时用户量小的时候发布，这样软件工程师和运维工程师不仅生活质量会严重下降，工作质量也会非常差。

还有一个更先进的做法就是把代码发布和功能启用进行架构拆分，先确保代码上线没有问题，再通过软开关来打开关闭某个功能，功能的打开和关闭就形成了一个新的发布生命周期。发布生命周期就被拆分为代码的发布生命周期和功能的发布生命周期，其中功能的发布生命周期是核心生命周期。这个做法可以快速地回退有问题的功能，而不用回退所有的代码，加速线上问题的处理。毕竟功能的启用关闭是内存的变更，不需要启停应用，比代码变更要快速很多。

在生产环境中，有些运维工程师和软件工程师有个坏习惯，就是喜欢在故障时慢慢探查原因，找到问题所在。在故障期间，时间是非常值钱的，业务的损失是按每秒钟来计算的。要树立一个非常重要的观念：

生产故障时要优先恢复用户的正常访问，而不是在生产环境探查问题的根本原因。

生产环境的问题，基本都是变更造成的问题，优先回退最近的变更，恢复业务的正常运行。也就意味着，所有的发布都必须要做好回退的方案，一般没有回退计划的发布是不允许执行的。

因生产问题发生回退后，一般会在生产环境保留有一到两台有问题的机器，不接入生产环境的流量，确保不影响用户的访问，留给软件工程师重现问题，因为只有重现了问题才能够定位并解决问题。如果新的软件只是部分上线，还只发布到 50%或者更少，最快的解决问题的办法，就是直接把这些新发布的机器全部从生产流量中拉出来。这样做比回退的速度更快，让故障对生产环境的影响时长达到最小。

在执行灰度发布时，每一个阶段发布前，都必须要检查之前所发布内容的监控和预警是否正常，否则不允许继续执行该阶段的发布。

一个新的软件发布后，必须建立全面的监控和预警规则，才能够移交给运维管理。在这之前软件工程师必须自行负责其新软件的监控和预警。

24.8　提升变更质量

之前提到拆分办公人员和用户所访问的环境，为了避免办公人员的访问和用户的访问互相影响，导致了生产环境的出现，形成了企业环境的架构拆分。

当办公环境和生产环境隔离之后，就会产生一个问题：生产环境只有一个，如何把办公环境生产的软件，发布到生产环境中？

如果直接发布的话，风险是非常高的。因为该软件从来没有在生产环境运行过，容易造成很严重的问题。所以往往会建设一个和生产环境一样架构的测试环境，也有独立的软件、硬件、电源和网络，用来模拟生产环境。该环境和生产环境的主要区别在于每个集群所包含的机器数量。生产环境的每个集群在该环境一般只有 1~2 台机器，因为测试环境流量很低，不需要多台，同时也为了节省成本。这个环境一般称为回归（Regression）测试环境，或者称为 Staging 环境。因此环境发生了进一步的架构分拆，在办公环境和生产环境之间形成了回归测试环境。

既然称为回归，重点就是测试新旧功能是否都能够正常地执行。回归测试一般用自动化来完成，因为大部分测试都是原有的功能。测试是一个独立的生命周期，有其自身的业务规律，这里不展开讨论。

因为生产环境只有一个，所以对应的回归测试环境也只能有一个。回归的基线是生产环境，要在这个基线上对新功能进行测试。而软件开发可以有多个版本并发进行，因此会有多个软件同时等待回归测试，甚至同一个软件会有多个版本等待回归测试，这就形成了回归测试环境争抢的问题。如果回归测试环境有多个的话，就失去了基线，模拟生产环境的目的就达不到了，也就是说一个回归测试环境不够用了。

为解决回归测试环境争抢的问题，就必须以回归测试环境为基线，建立多个功能测试环境，供测试人员做不同版本的新功能测试。该环境一般称为功能测试环境，因为主要是用来测试不同软件新增的功能的。这个环境同样也需要和回归测试环境以及办公环境隔离开，避免互相影响。因此，环境又一次地发生了架构分拆，形成了功能测试环境，处于办公环境和回归测试环境之间。

同样软件工程师开发时也是需要测试环境测试新功能的。为了解决软件工程师和测试人员对功能测试环境相冲突的问题，还需要建立一个开发测试环境，这

个测试环境也会和功能测试环境隔离，避免开发人员和测试人员共享，避免发生冲突。因此，环境再一次地发生了架构拆分，形成了开发测试环境，处于办公环境和功能测试环境之间。

此时就形成了软件在这几个环境中发布的推进管道（Pipeline）：开发测试环境–功能测试环境–回归测试环境–生产环境。所有的变更，包括软件和硬件的变更，都沿着这个管道依次的通过，最终到达生产环境。这就是一棵完整的环境架构树，用来隔离不同类型用户的访问，使得不同类型用户的访问通道互不干扰，同时通过流程来保证软件稳定地流入生产环境。

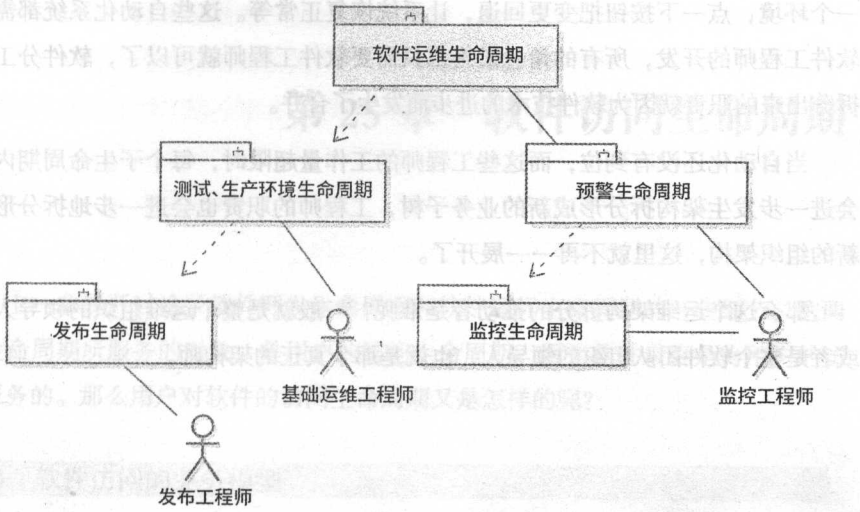
因为发布系统、监控系统和预警系统本身都是软件系统，所以也需要通过这几个推进的环节逐步地推进到生产环境。因为它们的部署，也是对系统的变更，也是有其开发测试环境–功能测试环境–回归测试环境–生产环境这个管道的。

如果是企业外购的系统，一般没有开发测试环境，但一定要有功能测试环境和回归测试环境，确保上线时不会和自研系统的测试和发布发生冲突。

另外一个影响变更质量的因素是手工操作。手工操作造成的失误是非常恐怖的。曾经在一个大型的互联网电商里面，一位数据库管理员（Database Administrator，DBA）在关闭测试数据库时，选择了错误的远程窗口输入命令，导致生产数据库被关闭，引起了重大的生产宕机事故。当一个屏幕中打开了很多远程机器窗口的时候，不同窗口的区别仅仅是 IP 不同，这种情况下是很难避免这种错误的。解决这类问题的唯一办法就是把发布变更的操作自动化，并且要让发布的动作在测试环境都测试通过没有问题之后，才能在生产环境发布。只有这样才能够避免人工的低级错误。

24.9 运维的架构拆分

经过上述的分析可以发现，运维生命周期发生了架构的拆分，形成了如下图所示的架构树：



在这个架构树中可以看到，软件运维生命周期的核心是预警生命周期。也就是说，保证软件运行生命周期的稳定是运维的核心工作。是否有预警表明被监控的软件是否运行稳定，预警是否消失则表明被监控的软件是否恢复正常。其他三个生命周期是为软件的稳定运行服务的，为非核心生命周期，都可以分工出去甚至外包。比如监控是为预警服务的，监控数据的记录可以交给第三方来做；环境的建立也可以交给第三方或者云服务；发布也可以通过第三方的工具来做。

唯有对预警的处理包含如何识别软件的不稳定状况和如何让软件恢复稳定这两个工作，必须要运维亲自来处理，不能交给别人来做。因为这两个工作和被监控软件所服务的业务生命周期有关，也和用户的访问生命周期有关，软件生命周期的核心是运行生命周期和用户的访问生命周期，而运行生命周期的目的就是保持业务生命周期的持续运行。这两个工作交给别人来做就意味着把业务也交给别人了，而别人无法通过业务来获益，此时权责不对等就产生了，因此交给别人是做不好这个工作的。只有企业自己的运维才能够把自己的业务维护和运营好，这样才是权责对等的。因此运维的核心生命周期就是预警生命周期，必须要运维自己来干，就好比每个人的饭要自己吃一样，这样才能够权责对等，保证内聚。

运维工程师角色相应地会被分拆为发布工程师、基础运维工程师和监控工程师等角色，甚至预警工程师也会被单独分拆出来。这些工程师的工作都可以自动化，人们只需要推动这些系统推进即可，比如点一下按钮发布，点一下按钮创建

一个环境，点一下按钮把变更回退，让系统恢复正常等。这些自动化系统都需要软件工程师的开发，所有的角色最终都只需要软件工程师就可以了，软件分工所拆分出来的职责就因为软件技术的进步而发生了合并。

当自动化还没有到位，而这些工程师的工作量超限时，每个子生命周期内部会进一步发生架构拆分形成新的业务子树，工程师的职责也会进一步地拆分形成新的组织架构，这里就不再一一展开了。

那么这个运维架构拆分的推动者是谁呢？一般就是整个运维组织的领导人，或者是整个软件团队组织的领导人，也就是那个真正的架构师。

第 25 章　软件访问生命周期

上一章展开讨论了软件开发生命周期和软件运行生命周期的一些细节。这两个生命周期所服务的对象，是用户的访问生命周期，因为软件最终是为用户的访问服务的。那么用户对软件的访问生命周期又是怎样的呢？

25.1　软件访问的业务模型

用户对软件的访问，核心在于“访问”二字，用英文来表达就是“Access”。很多人无法理解用户的访问是什么含义，访问用更通俗的话来说就是使用（或操作）。其实用户访问软件和人在现实生活中访问外部事物并没有什么不同，映射到现实生活中来理解可能会更加容易。

在人类早期，人们居无定所。为了保护自己的安全，人们学会了建立房屋作为自己的领地。人对房屋的使用就是对房屋的访问。房屋建好之后，人们为了使用房屋，会留有门来进出，并且只有门能够访问房屋，人们访问自己房屋的路径就增加了一道门。这是人们对房屋的访问生命周期所做的架构分拆，拆分出了门的生命周期，门的开合就是一个访问通道开闭的生命周期。房屋和门因此变成了一个新的社会分工，人类都需要房屋和门来为自己服务，房和门就形成了一个模式，或者一种技术。

随着人类的增多，地球的区域资源会发生不足，因此在区域内会发生资源争抢，形成冲突。资源富有地区的人们为了保护自己，就联合起来给自己居住的地区加强保护，建立了一个更大的城墙，留有城门来进出，并设立军队来保卫这个城墙和城门。因此人们访问自己财产的生命周期又发生了分拆，拆分出了城门进出的生命周期。城墙和城门也变成了一个模式或技术，大家竞相模仿。

随着人数的进一步增多，慢慢就形成了城市、省和国家。从外部来的访问路径越来越长，分工越来越多，社会也就开始越来越复杂。

在形成一个一个的保护区之后，访问保护区的人就分为两类：一类是人们访问自己财产的，属于保护区内财产的拥有者；一类是观光或购物，甚至是来攻击或掠夺的，不属于保护区内财产的拥有者。这个时候保护区大门的任务就变成过滤访问者，识别自己人，同时严防来攻击的人，毕竟保护区能提供服务的资源有限。同时，通过开放给观光或者购物的人，可以吸引外部资源进入，也可以留住优秀的人，添加保护区的活力。

在计算机和软件领域，人们对软件的访问和人类对外部资源的访问有什么不一样吗？

为了保障软件的运行生命周期，运维会设立一个“墙”来隔离出生产环境，使得生产环境的变更只有一个物理入口和一个虚拟的网络入口。而生产环境的目标则是为用户提供访问，因此会有一个入口专门提供给用户访问。用户的访问也是生产环境变更的来源，但这是被动变更，不在主动变更的范围。

因此，生产环境就好比是一个城池，一个保护区，用户的访问则是进出这个城池。为了保护这个城池的安全，城墙是必须有的，确保用户访问城池只能通过城池的大门进出。对于进出访问的用户，也要区分是自己人，还是来观光购物的人，对于举止可疑的人，还要进行盘查，阻止恶意的人进入。用户对生产环境的访问和现实生活中对城池的访问是一样的。

为什么会想到要和城池来进行映射呢？人类发明计算机和软件，实际上就是为了模拟人类自己，软件多了就成了模拟的人类社会。当软件建立以后，虚拟人生活在计算机中。由于虚拟人不具备意志，因此真实的人在现实世界通过鼠标键盘来访问计算机，控制软件社会中虚拟的自己，推进虚拟人的生命周期变化，就好比木偶戏中人控制木偶一样，只不过在这里是通过鼠标控制计算机里的木偶人而已。而为了保护软件的运行，运维会对运行这些软件的计算机形成保护区，即建立城墙。这和现实生活中访问另一个群体没有什么区别，做法是一样的。人们总会不知不觉地、自然而然地把自己最熟悉的方式映射到软件中。因此我们要理解现实生活，才能够更好地理解软件的社会，才能够做好软件，在这里不得不一再强调这点。

历史总是惊人的相似，人类的发展所经历的过程，也在软件社会中重复发生。比如攻击、战争等。为了防止攻击，生产环境还需要设立 DMZ(Demilitarized Zone),

也就是所谓的“非军事区”，这也是来自于现实生活中的战争实践。非军事区是一个军事用语，就是在交战双方之间设立一个区域，进入该区域的条件就是不能携带武器。在互联网时代，DMZ 区域主要是用来隔离生产环境的内部网络，在内网和互联网之间设立 DMZ 区域，形成用户访问的架构拆分，使得用户无法直接攻击内网区域。

设立了 DMZ 之后，用户对内网的访问就由 DMZ 来代理进行，此时不允许用户自行进入内网访问。在现实世界，任何人进入 DMZ 的区域都需要解除武装。在虚拟世界无法做到像现实世界那样解除用户的武装，实现的方法是提供一个代理人，由代理人代替用户来访问。因为代理人是软件内部所认可的自己人，所能做的事都是得到认可的、安全的，相当于解除了用户的武装，避免了被攻击。部署在 DMZ 中的代理人拿着用户的信息，把用户的请求转达到内网的服务器，代替用户完成内网的访问，再把访问的结果交给用户。

用户进入城池内部时，还需要不同的路标指引用户，帮助用户找到访问的目的地，这就是路由。路由负责把用户的请求传达到相应的服务器，获取所需的服务。

由于软件的部署架构是一棵树，因此用户对软件的访问从数学上来说，实际上就是对部署架构树的一次遍历。

用户的访问也是有生命周期的，从用户发出请求开始，到获得所需要的服务为止。用户每完成一个业务生命周期，可能会引起多个访问生命周期。应用服务器要跟踪并记录用户的每个访问生命周期，就如现实世界中访问一个城池一样。为了安全要对每个地址的访问都进行登记记录，以便后续的数据分析、监控和预警。

以上就是访问的业务模型，和现实生活的做法是保持一致的。大家应该结合现实生活去理解访问生命周期，这样就能够明白运维的做法和理由。当自己作为运维或安全人员面对访问问题的时候，也知道怎么去思考了。

25.2　软件访问路径的架构拆分

最早的软件开发中，软件工程师面对的是硬件设备，是直接针对硬件进行开发的。因此，用户在访问软件时，经过这样的路径：用户→目标软件→硬件。用户操作软件，软件依赖硬件而运行。

随着软件工程师数量的增多，他们中越来越多的人面临着同样的问题：如何管理硬件的资源？因此，一部分软件工程师转而去专门开发软件来管理硬件的资源，形成了软件的分工，于是操作系统逐渐地出现，把硬件资源统一管理起来，并给开发人员提供访问资源的接口。用户访问路径发生了架构拆分，在目标软件和硬件之间形成了操作系统生命周期，变成了：用户→目标软件→操作系统→硬件。操作系统的出现，让软件工程师不再需要直接和硬件打交道，软件开发的门槛下降了，开发软件也变得更加容易。

计算机在早期还是非常昂贵的，贵的东西通过提高使用率可以降低成本。随着网络的兴起，软件被拆分为两部分，一部分在服务端运行，一部分在客户端运行，客户端可以用非常简单的终端连上服务器，使得同一台计算机可以被多人同时共享使用。用户对软件的访问通道进一步发生了架构分拆，增加了客户端软件、网络等生命周期。这通常被称为 C/S 架构（Client/Server）。因此用户的访问路径变成：用户→客户端→网络→目标软件→操作系统→硬件。

随着服务端编程复杂度地上升，服务端软件发生了架构分拆，软件分拆为应用服务器和运行在应用服务器上的软件，应用服务器的生命周期就出现了。应用服务器，顾名思义，就是为应用提供一个容器，负责生成端口，和用户的的请求交互，并把请求传递给应用服务器所容纳的软件来处理业务。因此用户的访问路径变成：用户→客户端→网络→目标软件→应用服务器→操作系统→硬件。

随着云服务的兴起，硬件的利用率问题慢慢浮上台面，虚拟化、容器化也成为了一个方向，在操作系统前形成了一个新的架构分拆，形成了容器的生命周期。用户的访问路径变成：用户→客户端→网络→目标软件→应用服务器→容器→操作系统→硬件，或者：用户→客户端→网络→目标软件→应用服务器→操作系统→虚拟化→硬件。

从这个发展路径上来看，用户的访问生命周期在不断地被拆分，分工在不断地增加。同时软件本身也在不断地拆分，从一个软件做所有的事情，拆分为软件和操作系统，业务软件不再需要直接和硬件打交道了，和硬件打交道变成了其他软件的业务，比如操作系统等。接着软件拆分出了应用服务器，软件不用再直接和网络打交道了。随着 MVC 框架的兴起，软件得以真正的专注于所服务的业务需求。软件访问路径的拆分导致软件的分工越来越多，也越来越细，不同的分工形

成了不同的业务，这些不同业务又被各种软件所模拟，形成了各种不同的软件技术。

可以得出结论，访问路径上的业务，都是计算机和软件自身的业务，是为降低软件的开发难度而沿着访问生命周期做架构拆分而形成的。

为了仍然能够让用户的访问到达软件，访问生命周期的架构拆分形成了不同的子生命周期，形成了计算机和软件自身的各种业务。目的仍然是为了让软件能够更好的服务于模拟现实世界的业务。也就是说：

计算机和软件自身的业务都可以归结为访问通道型的业务。

软件工程师和运维工程师大都忽略了计算机和软件本身的业务和人类历史发展的关系。要真正掌握计算机技术，仅仅钻研技术是远远不够的，必须先要懂得，计算机和软件本身的业务都是对用户的访问生命周期做架构分拆而形成的。

25.3　大规模软件访问的架构拆分

当一个餐馆非常受欢迎，大家都排着队去吃的时候，这个餐馆就会扩张。一种扩张方式是增加座位，这个方式受限于厨房的大小，厨师的人数，以及餐厅的空间容量；另一个扩张办法就是开一个分店，还能方便分店附近的客户就近用餐。再比如随着一个城市人口逐渐地增多，会建造越来越多的房屋，用来容纳越来越多的居民。当地不够用的时候，房屋则会造得越来越高。

这就是两种基本的增长模式：以复制自己的方式增长的，叫做 Scale-Out，或者叫做横向扩展；以增大个体能力的方式增长的，叫做 Scale-Up，或者叫做纵向扩展。

同样，当软件的用户量越来越大的时候，一台机器势必会造成访问的拥堵。另一个问题是，只有一台机器的时候，万一这台机器出故障了，那么软件运行生命周期就终止了。这也被称作单点故障（Single Point of Failure）。单点本身是没有问题的，比如餐馆有时候还歇业呢，人也不能 7*24 小时不停运转。但是如果餐馆老板有虚拟的人可以帮他 7*24 小时不停地干，他就舍不得歇业了。为了最大化利用好资源，自然就希望永远别停。在这个情形下，单点就成了一个痛点。

纵向扩展可以增强访问能力，但是无法解决单点问题，而横向扩展既能增强

访问能力，又能解决单点问题。似乎横向扩展是软件访问容量能力增长的首选。

现实生活中如果增加一个分店，需要重新租地方、招厨师、招服务员等，成本非常得高。当店老板野心不是太大的时候，一般不会选择去横向扩张，宁愿客户排队，或者增加一些座位。

但是在虚拟世界，横向扩张只需要增加机器即可，似乎成本更低。但是在早期，计算机价格很贵，因此横向扩展成本也非常高。随着计算机行业地发展，计算机的成本变的非常便宜，同时网络的能力也越来越强，横向扩展的成本变得非常低。横向扩展就成为了软件访问能力扩展的首选。

由此可见，扩张没有最好的方式，选择横向扩张还是纵向扩张都要根据当时社会的技术能力水平来判断。判断标准是对比两种扩张方式的成本和收益，同时也要看业务拥有者本身的扩张意愿。

在软件行业，由于计算机的成本变得很便宜，提升软件访问能力的扩张方式基本都是采用横向扩展。因此集群（Cluster）的概念顺应而生，虚拟世界迎来了翻天覆地的变化。集群的生命周期被拆分出来，形成了新的独立业务。

25.4 集群

什么是集群呢？集群就是装有相同软件，具备同样功能的一组计算机，组织在一起共同服务于客户的访问。结果就是软件把自己的复制了多套，像孙悟空七十二变一样变出了很多个同样的自己，此时虚拟人的优势就体现出来了。这和人们一个人活干不过来，招很多人帮自己干活是一样的道理，但是成本更低。招人来还需要培训，软件只需要复制一下就可以让新的计算机具备同样的技术和能力，所付出的代价只有新增的计算机和网络成本，远比人便宜。

在集群出现之前，要提升计算机的计算能力，只能升级 CPU、内存等，受限于计算机的体系架构，增长非常有限。或者购买小型机、大型机等，价格又非常昂贵。计算机网络出现后，通过网络来连接不同的计算机，再通过形成集群变成了一个更大的计算机。集群的价格对于小型机、大型机便宜太多，也比升级机器的硬件的方式更容易扩展。因为计算机本身计算能力的增长受限于当时计算机硬件技术的发展。而以集群的方式横向扩展，可以达到远远超过单个计算机本身能力的效果，受计算机本身技术发展的限制就不那么明显了。

应用集群后，所有的用户不再只访问一台机器，变成了用户分散访问不同的机器，但还是访问同一个软件。因为集群内的机器功能都是一样的，相当于增加了访问软件的通道，使得软件的访问容量得到了增长。

采用集群后会形成一个新的问题：一个用户只需要访问一台机器上的软件，但有这么多装有相同软件的机器，用户应该选择哪台机器来访问呢？要解决这个问题，势必不能让用户在访问时自己来选，需要一个机制来帮助用户来选择某台机器。

现实世界里当人们访问某个目的地，比如走到路口，经常是需要路牌来指示往哪个方向到达哪个目的地的。同样对集群内计算机的访问也模拟了这一点，就是路由（Router）。只要在用户访问到达集群前端时做一个路由，把每个用户的请求按照预设的规则转发到集群中的某台机器，即可让用户的访问自动到达某台机器上的软件。转发预设的规则还需要考虑某台机器是否过忙，否则把用户的请求转发到过载的机器上反而影响用户的访问。

因此集群的路由主要考虑的因素是集群内机器的负载均衡问题，为了实现对集群内机器负载的感知，路由必须定时地检查集群内机器的健康和剩余访问容量。路由的引入使得用户并不知道自己访问的是一个集群，因为每次访问都只访问一台机器，集群对用户是透明的。用户不知道软件被复制了多份，用户上一次所访问的机器和当前所访问的机器有可能不同，但是用户无法感知到，就好像是访问同一个软件一样。

集群出现后，用户的访问路径又做了一次架构分拆，在软件所部署的计算机之前增加了一个访问路由，这个路由负责把用户的请求按照转发规则转发给集群内的某台计算机。转发的规则有很多种，比如按照访问的权重来路由，随机路由、散列路由等，各有各的优势和缺点。但是无论哪种方式，都是为了让集群内的计算机都能够服务到用户，避免某些计算机负载过重，而有些计算机却吃不饱。

集群虽然很好，但是要实现集群，对软件本身则有一个限制：要能够在不同的计算机之间保持用户的状态。比如用户本次访问的机器和上次不同，这个机器就不知道用户是谁，从而无法给用户服务，就要求用户再次登录。在这个情况下，用户感知到了集群的存在，集群就失去了意义。要做到集群对用户透明，软件本身还需要在用户的访问路径上做一个架构拆分，使得同一个集群的不同机器之间，

能够共享用户的状态。只有做到这一点，才能够真正地让集群对用户透明。当然还有一种路由转发规则是按照用户的来源固定给某台机器来服务，确保单个用户上次访问的目标计算机和当前要访问的目标计算机是同一台来避免这个问题。实现方式是按照访问来源地址对集群内机器数量做散列，这样某个来源地址所访问的机器就是固定的。但是这个方式也会带来很多其他的问题，这里就不展开讨论了。

当一台机器变成了一个集群，就好比是现实社会中一个村庄变成了一个城市，可以容纳更多的人访问。可是这个世界有超过 70 亿的人，如果都来访问一个城市的话，这个城市是承受不了的。当然，在现实社会这样情况是不会发生的，因为有国家的边界限制、交通通道的限制等。但是在虚拟世界，这个现象已经正在发生，如很多网站已经超过 10 亿的用户数。这已经体现出虚拟世界的强大了。

另一方面，虽然光和电的传输速度已经非常快，但是地域空间距离对虚拟世界的影响还是显现了出来。在国内，从南方访问北方的网站，延时超过 50ms。从中国访问美国的网站，通过横跨太平洋的光缆，响应时间超过 100ms。更远的地方，响应时间会更长。这个响应时长，偶尔使用问题不大，一旦长期使用，每次访问都会延时这么长，就难以接受了。因此多数据中心的部署就出现了。

从这里我们也可以看出，虽然虚拟世界可以从某种程度摆脱地理位置的影响，可是地理位置，也就是空间，从来没有放弃影响我们的机会，始终潜移默化、不知不觉地围绕在我们周围。根本原因就在于，虚拟世界不具备人的意志，其生命周期活动的推动还是要靠现实世界的人类，而人类的身体对外部事物的访问要受物理空间规律的限制。

25.5 数据中心

什么是数据中心（Data Center）呢？前面说集群像是一个城市，多数据中心可以看成是集群的集群，装有同一个软件的集群会同时部署在多个不同的数据中心，集群在不同的数据中心各复制了一份。多数据中心就相当于现实世界中的多个国家。但和现实世界的区别就在于，一个国家的城市主要局限于这个国家所处的地理位置，而数据中心可以分布在全世界，摆脱了一个国家地理位置对虚拟世界的影响，让全世界的人都能够就近访问数据中心，访问虚拟的世界，降低空间地理

位置对人类访问软件的影响。于此同时，也可以获得另一个好处。当一个数据中心出问题了，比如战争、地质灾害等，其他的数据中心可以接管，可以让软件保持持续运行，获得很长的运行生命周期，不会对用户的访问造成影响。因此产生了数据中心的生命周期，新的业务又被剥离出来了，用户访问软件的生命周期又发生了新的拆分。

要做到用户在不同数据中心访问同一个软件，就必须在不同数据中心的前端放置一个路由，类似于前述的集群做法，为部署了同一个软件但位于不同数据中心的各个集群做路由。用户访问软件的路径又做了一次架构拆分，增加了数据中心的路由生命周期。

数据中心前置路由的路由策略和集群的路由策略有不同之处。集群的路由策略保障的是集群内机器访问的均衡，而数据中心前置路由的路由策略则是把数据中心所覆盖地区的用户访问归属到相应的数据中心。这就意味着在建数据中心时，要考虑的实际上是物理空间维度上用户的覆盖面，避免数据中心过小而用户过多，导致数据中心之间的负载不均衡。这就非常像国家对其地理空间范围内人民的管理了，也意味着用户在虚拟空间的分布有了现实的参照，形成了数据中心本身的业务领域，有其自身的生命周期规律。毕竟用户的划分本身就是一个空间维度的组织，数据中心是一个虚拟的地理维度，组织其范围内的虚拟人，也是对现实社会人类组织的一个模拟。

有了多数据中心，一旦某个数据中心不能工作，最多影响的是归属于该数据中心用户的访问。相当于现实世界中某个国家发生了动乱，但是对其他国家的影响有限。有了多数据中心，数据的备份也变得相对安全，地理空间发生的动乱灾害，在另一地会比较安全，除非全球同时出问题。这其实就把软件的命运和整个地球的命运绑定在了一起，因为地球生命周期的时间跨度是远远超过人类社会生命周期时间跨度的，相对于人类社会更安全。

同一个软件部署在多个数据中心的不同集群上，一般会重新定义一个名字，叫做池（Pool）。比如有两个数据中心 shanghai-dc 和 beijing-dc，都分别有订单集群，它们的名字会是 order-shanghai-dc 和 order-beijing-dc，但是它们都属于订单的集群，会有一个统一的名字，叫做订单池（Order Pool），相当于是一个集群池，方便和集群区分开。多数据中心出现后，软件发布的目标不再是针对集群，而是针

对池。集群是物理的，池可以认为是逻辑上的集群，多个数据中心里部署了同一个软件的计算机都属于同一个池。

多数据中心的出现，对变更的发布也会造成影响。一个池会同时部署在多个数据中心，发布变更的时候不能把一个数据中心的集群全部关闭。比如上例中订单池在两个数据中心分别有一个集群，各 100 台机器。在软件发布到 50%的时候，不能让某一个数据中心的 100 台机器全部下线，这会让一个数据中心的订单集群变成完全不可用，这是不可接受的。发布 50%的时候应该是让两个数据中心各发布 50%。

当一个用户从一个数据中心迁移到另一个数据中心时怎么办呢？既然数据中心相当于一个国家，那么可以参考现实生活中不同国家的移民。要做到这一点，只要按照用户维度存储数据，用户的迁移就非常容易了。还有一个办法就是通过数据同步保证两个数据中心的数据一致，现在网络和存储都很便宜，代价也不算高。同步的延时问题可以在用户的访问端来解决，比如在同步完成后再给用户切换数据中心。

由上文的讨论可以看到，用户的访问生命周期至关重要。在计算机发展的过程中，为了提升这一点，发生了很多的架构分拆，让用户对计算机软件的访问变得越来越简单，软件开发的分工也因此变得越来越细，软件开发也变得越来越容易，运维的能力也同时得到了提升，大规模的软件应用变成了现实。可以想见，未来用户对软件的访问还会进一步地拆分，用户对软件的访问也会变得更加的容易和自然。

很多企业专门设立了系统架构师这样一个角色，整天在研究负载均衡、网关防火墙等技术。其实所谓的系统设计，目的就是为了处理好用户的访问生命周期。要做好这个工作，就必须深刻理解用户的访问生命周期，也要理解访问生命周期是如何推动并形成硬件的拆分和部署以及网络的拆分和部署的。系统架构师的核心在于执行，如果只具备系统架构师的头衔，但是没有执行能力的话，仍然不是真正的架构师。具备执行能力的才是真正的系统架构师，也往往是运维团队组织的领导者。

很多架构师和软件工程师非常专注于技术，以致于不重视各种技术背后的业务，导致学习技术的过程非常艰难，始终无法把技术和现实生活对应起来，汲取

不到力量。计算机相关的技术，围绕的核心点始终是如何让用户能够访问到软件，以及如何能够更加容易的访问到软件。所有访问相关的技术都是来源于现实生活的。学习人类历史，会帮助理解架构分拆背后的原因，也可以从中吸取营养。不理解人类社会本身，计算机和软件相关的技术是很难掌握好、应用好的，反而容易被技术所限制，被技术所控制，最后变成技术的奴隶。

第 26 章　软件架构和大数据

大数据（Big Data）的概念提出有些年头了，这个概念也是一直没有一个统一的定义，每个人都有自己的理解。大家都把专注点放在“大”上，每个人在自己的立场，对“大”都有各自不同的阐述。比如有观点认为，把企业所有的数据都收集起来，这就是“大”，也就是“全”的意思。也有很多人反对“大”，因为数据集太大，反而导致有用的数据被淹没，增加了提取有效数据的困难。

26.1　什么是大数据

那么究竟什么才是大数据呢？

在大数据提出之前，对于大量数据的处理已经有很多现成的技术，比如数据仓库（Data Warehouse）、数据挖掘（Data Mining）等。可以注意到，这些技术的基石都是关系型数据库。因此，从大数据这个概念产生的缘由来看，原因之一可能是因为数据量已经超过了关系型数据库的处理能力。另一个原因可能是原有技术对数据的处理时效太长，都是以天计算的，也就是 T + 1 的方式，比如当天抽取数据，第二天才出分析的结果。

在瞬息万变的市场中，T + 1 的方式是很难让商业决策者满意的。企业的决策人员需要更全的数据、更快的处理速度，来快速得到分析结果，帮助做商业判断。而这恰恰是关系型数据库的弱点。为了克服这个弱点，出现了很多针对大数据集处理的工具和技术，超越了关系型数据库的限制。这些对大数据量的数据进行快速处理的工具和技术被统称为“大数据”。

因此，大数据并非是某一种具体的数据，也并非指数据是否大，而是相对于以前的数据处理方式而言的，是对关系型数据库处理方式的颠覆。毕竟关系型数据库统治了数据处理领域这么多年，打破常规需要巨大的勇气和自信，这就是技术人的优势。所以大数据其实说的是新的针对大量数据的处理技术，并非“大数

据”这个概念表面文字那样是说“数据”本身。可能也是这一层原因，让大家对大数据都有自己的理解。这也不是坏事，可以促进大家的思考。

26.2　如何做好大数据

那么怎么样才能做好大数据呢？

既然大数据指的还是大数据集的处理技术，那么技术所服务的还是业务，这个业务就是“数据”。

当数据集很全时，确实会对数据提取和处理造成干扰，但这只是现象。事实上所有的数据都是有用的。背后的原因是做分析的人对数据不理解，不知道这些数据是什么含义。这才是真正让人非常担忧的，技术的进步无法解决这个问题。因为这是人的问题，业务的问题，而不是技术的问题。

要做好大数据，真正的焦点不在“大”，而在“数据”本身。因此我们要先真正的理解数据，才能够处理好数据，让数据产生价值。

而要理解数据，则要先知道，数据只是行为的状态、行为的结果；

而要理解行为，则要先理解产生行为的业务；

而要理解业务，则要先理解业务的目标；

而要理解业务的目标，则先要明白业务的主体；

业务的主体明确了，就可以理解业务的目标；

理解了业务的目标，就可以知道业务的生命周期；

知道了业务的生命周期，就可以知道业务生命周期的架构拆分；

知道了业务的架构拆分，就可以知道业务的组织架构；

知道了业务的组织架构，就可以知道业务的分工；

知道了业务的分工，就可以知道业务中每个角色的生命周期；

知道了每个角色的生命周期，就知道了生命周期中的关键行为。在面向对象的开发方式中，每个生命周期都是一个对象；

知道了关键行为，就可以知道不同生命周期推进的进程中所产生的数据积累；

知道了业务生命周期进程中的数据积累就理解了数据。

没有软件时，现实生活中要收集这些业务数据是非常困难的。因为大自然不会留下行为的痕迹，只会留下当前的结果，已发生行为产生的数据随风而逝。但人类需要这些数据，就要每时每刻地记录当前数据，而要记录这些变化会耗费极大的人力。比如要收集一次足球比赛的数据，如每个球员的触球、抢断、奔跑、射门等，人们不得不紧盯着画面，对每个队员，每个动作做人工的记录，成本非常的高。

当把业务在软件中虚拟出来后，这些业务数据的变化，每一步都可以用软件记录下来，形成业务日志。在软件中做这些记录非常容易，成本也非常低。有了大数据这个工具，分析这些业务数据就可以得到不同生命周期的变化过程，得到不同业务决策对用户行为的影响，可以帮助判断不同生命周期的运转是否良好以及整体业务运营是否和预想的一致等。这其实就是业务的监控和预警，或者叫做业务的运营。

从另外一个角度来看，只要对实时度和数据集大小的要求不高，关系型数据库还是可以很好地完成任务的。理解了业务，一样可以把业务的监控和预警做好，并不一定要采用大数据技术。毕竟大数据对人，对基础工具的要求都完全不一样，还是有一定的成本的。真正的重点在于理解业务，而不是技术本身。有些场景下关系型数据库还是有自己的优势的。

所以，业务生命周期的架构拆分，是理解业务数据的基础，也是做数据分析的技术，还是大数据或数据仓库的基础。

26.3 软件大数据

软件是负责虚拟化业务的，因此软件本身也成为了一个独特的业务。为了服务好业务的生命周期模拟，软件本身的数据分析也就非常的重要。

和业务大数据分析一样，要做好软件大数据也需要梳理清楚软件生命周期架构的拆分。软件的生命周期被拆分为开发生命周期、运行生命周期和访问生命周期，形成树状的架构，核心是运行生命周期和访问生命周期。

用户对软件的访问生命周期体现了业务的访问行为，也同时体现了软件本身

的访问行为，因此访问日志（Access Log）可以被用作业务数据分析和软件本身的业务分析。所有的服务器都带有访问日志，不需要额外的开发成本。

对访问生命周期的架构拆分，形成了路由、数据中心、集群和网络等各种设备，这些拆分出来的生命周期都有各自的访问日志和业务数据。这些数据的搜集被用作软件和硬件系统的监控及预警。当服务器和设备足够多的时候，比如几十万台甚至更多，就是典型的大数据应用场景。

当业务需要从大量的软件访问日志中抽取用户的业务行为时，也是典型的大数据应用场景。

对于开发生命周期，每个新的软件版本的开发生命周期也有自己的拆分，拆分出来的生命周期同样会产生数据的积累。通过收集这些数据，可以判断开发生命周期的运作是否良好，为版本的安排提供量化的依据。

每个变更版本的发布，对访问生命周期的影响如何，会不会导致访问质量的下降，新上线的业务是否能达到相应的业务目标？这些都需要大量数据处理支持，都来源于架构的拆分所形成的生命周期，以及这些生命周期所产生的数据。

得到的数据只是一个表象，简单的收集数据并不能解决问题，哪怕采用大数据技术也是一样。不理解这些架构的拆分，不理解这些拆分出来的生命周期的积累，是无法理解数据的。即便对数据很了解，如果不知道这些生命周期是如何拆分出来的，也就不能够判断它们相互之间的关系，最终无法形成有效的决策。因此，架构对于大数据处理至关重要。

从上文的分析也可以看出，软件的大数据分为两部分。一部分是业务的监控和预警，也就是业务生命周期和拆分之后的各子生命周期之间的联动变化规律。这部分往往被称作业务的运营，由业务方来执行。还有一部分是软件本身的监控和预警，也就是软件生命周期和分拆之后的各子生命周期之间的联动变化规律。这部分往往被称作运维，由软件的运维团队来执行。

由于业务运营的数据往往来源于运维团队，所以业务的运营少不了运维的支持和参与，也就是说业务的运营会访问运维团队。针对不同的业务团队，运维团队内部还要对此进行分工，形成不同的访问通道，避免不同的业务方互相干扰，从而对自己产生不必要的沟通成本。也就是说，运维也要针对业务团队对运维团

队的访问，做访问通道的隔离。为了做到对业务方的支持，运维还需要把软件产生的监控数据做业务数据和软件数据的分离，因为业务运行数据往往会和软件运行数据混合在一起，比如访问日志就是典型的例子。

第 27 章　软件架构和建筑架构

在软件行业说到架构，大家都默认为软件架构。但是在软件行业之外，大家谈到架构，往往是说建筑架构，可见建筑架构很深入人心。即便是软件架构本身，也在不断地从建筑架构成熟的体系中吸取营养。

要探讨软件架构，建筑架构自然是一个绕不过去的坎。本章就尝试讨论一下软件架构和建筑架构的异同，与建筑架构的对比或许还可以加深对软件架构的认识。毕竟“他山之石，可以攻玉”。

27.1　软件架构和建筑架构的目标之异同

首先，软件架构和建筑架构都是为人服务的，两者在这一点上是一致的，都是为了解决人类自身生命周期的问题，对世界所做的一个拆分。并且拆分的结果，都是对人类形成了分工。拆分的目的，也都是为了延长人类的生命周期，提升人类生命周期活动的推进质量。

在没有建筑的时候，人类是非常脆弱的，也是非常渺小的。来自美国的迈克尔·史蒂文斯（Michael Stevens）利用计算机算法，将全球 72 亿人堆叠起来，放入科罗拉多大峡谷当中。72 亿人其实总共只有这么一小堆，一个 788 米高的小堆。

人没有最强壮的身体，无法抵御野兽的攻击；人不具备最快的奔跑速度；人也没有足够的耐力；人体表没有保温的皮毛，地球的四季气候变化，对人的杀伤力巨大。而人偏偏具备专心做事的能力，不像其他动物随时对外界的威胁保持警觉。人在专心做一件事的时候，往往也是最脆弱的时候，非常容易受到攻击。比如人在睡觉的时候，是非常容易被攻击受到伤害的。人如果披上一个铠甲就能形成一个仅仅容纳身体的独占空间，可以帮助防护攻击，但是铠甲同时又限制了人的活动能力。人类创造一个更大的独占空间自然就成了一个目标，这就是建筑。

建筑架构的目标，是为了人类能在地球上拥有一个独占的空间，但又不至于

与世界隔绝，仍然能够保持与自然界的沟通，并且尽量让人类掌握这个沟通的主导权。要形成建筑，先要遵从地球本身的规律，即地球的引力，因为建筑必须要从地表形成空间。

在地表形成空间的方式有两种：一是从地面向上形成空间，二是从地面向下形成空间。最终建筑都是要占有一部分地球表面的空间。因为建筑是为人服务的，所以建筑要考虑人的特点。比如人喜暖怕湿等，这些因素决定了大部分建筑要从地表往上发展。有些地区，比如北方，因为地表很冷，建筑反而建在地下并往下发展。因为这些地区地下空气中水分很少，土地可以帮助保温，适合人居住，建造的成本也低。

建筑要建在地表上，就需要考虑几个外在因素：地表要稳定，要朝阳，要考虑风的因素，必须要离水源近等。这些都是在人类生命周期活动中必须有的要素。

为了方便人的生活，建筑的内部还要考虑人的个体在建筑内的生命周期，如吃、喝、拉、撒、睡等。家庭是有多个人的，于是要对建筑内的空间再做切分，将人的生命周期活动限制在不同的空间执行，错开每个人的生命周期活动对其他人的影响，形成了餐厅、卫生间、厨房和卧室等典型的空间切分，形成了不同的切分模式。空间被切分后，为了让每个人能够在这些空间之间进行沟通和走动，还需要对访问生命周期做架构拆分，此时就形成了门等设施。人还要呼吸空气才能生活，因此还要对空气的访问生命周期进行管理、拆分，形成窗等设施，等等之类。这些不同的访问也要各自有独立的访问通道，避免他们之间的访问互相影响。比如空气访问通道就不能和人类的访问通道共用，否则一旦晚上睡觉关门了，人类呼吸空气就成了问题。

为了做好这些空间切分，就需要寻找或者发明隔离这些空间的材料，以及隔离空间所需要的技术，以便实施对空间的架构拆分，这就是建筑材料和建筑的技术。建筑的材料要克服地球的重力、地球本身运动和大气运动带来的变化，以及地球气候周期性变化对材料的损耗等因素，还要考虑建筑和周围环境的融合，和周围空间的沟通，比如道路、风向、阳光等。所以，建筑架构师往往让建筑和自然融为一体，把保持建筑的长久生命力做为自己的职业生涯目标，也就是尽可能地延长建筑本身生命周期的时间跨度。

空间架构切分设计好了之后，还需要实施空间架构的切分才算是真正完成了

架构，这就是为什么说架构的核心是实施。如果设计一个“空中楼阁”，无法实施，这就不算架构。建筑实施的时候，主要考虑的就是通过力学计算，用不同材料来切分空间，实施空间的拆分。很多人认为架构就是一个框架或架子的原因就在于此，要知道框架和架子的目的仅仅是为了实现架构对空间的切分，属于实施的技术，并非架构本身。

材料的选择必须考虑地球表面自然环境的影响，如冷热、风雨、地质变动等，材料自身的生命周期跨度要足够得长，这是建筑长久存在的一个前提。因此建筑所用的材料大都是从大自然获取的，并且是已经久经大自然变化考验的材料，比如木头、石头等。建筑空间的切分要真正地提升居住人的生活质量，让人们居住其中获得一个稳定的生命周期。人们住的舒服了，才能够让建筑获得长久的生命力。因为只有人们愿意住进去，形成一个健康的访问生命周期循环，建筑才能够得到很好的养护，及时替换损坏了的材料，也就是建筑的运维。

建筑还未建造时，可以通过小的模型把建筑的空间划分表达出来，确定设计本身是否满足客户，也就是居住或拥有人的目的。经历了上千年的发展之后，建筑的设计以及实施都变得非常成熟。在近现代飞速发展的材料和工艺的帮助下，建筑不断地获得巨大的生命力。

而软件的目标关注的是把人们所做的事情模拟出来，把人从重复的劳动中解脱出来。世界是周期变化的，就意味着软件一旦把这些周期模拟出来、管理起来，人们只需要按照自己的需要，推动这些生命周期为自己服务即可，不需要亲自去执行这个周期。也就是说，软件所关心的是人类生命周期的切分，把人类自身生命周期活动以外的非核心生命周期用软件来模拟实现，让人类可以有更多的时间来处理自己的生命周期，提升生命质量。

也就是说，建筑创造的是一个物理的世界，软件创造的是一个虚拟的世界。甚至建筑本身也是人类的一个活动，也在大量地采用软件来模拟，帮助建造更好的建筑。比如力学分析和空间切分模拟可以利用软件来更容易地进行，材料分析和处理也可以更加精确，甚至可以做到建筑在地球整个大环境下的变化模拟，估算建筑的寿命。

建筑和软件，都有建造的生命周期和运营的生命周期的拆分，都需要先建设起来才能够运营。建筑的空间最终是要交给用户来使用的，软件的目的一样也是

最终让用户来访问的。不同之处在于，建筑的设计建造往往是由建筑公司来做的，几乎没有人会因为要造一栋建筑，自己养一个建筑设计施工团队。而如果企业需要建造软件的话，往往要自己养一个软件研发团队，用来负责软件的研发和运营。建筑的运营甚至可以完全交给物业公司，软件目前也还做不到这一点，必须要自己来运营。

另外建筑可以出租给其他的用户来使用，而软件的出租则比较难，只有以娱乐为目的的软件，或者标准化的软件相对容易。这是因为建筑的目标是为了占据一个空间，空间可以分为空间的使用权和所有权，建筑本身只是划分空间的工具，而不是空间本身。因为建筑的出现，空间的独占性成为了现实，出租的实际上是空间，而不是建筑。但建筑空间划分又和人的核心生命周期活动相关，是非常标准化的，可选择的也很多。在几千年的历史中，人类的核心生命周期活动相对稳定，还是吃、喝、拉、撒等。即使是千年前的房屋，现代人仍然能够正常居住。因此出租给其他人使用是比较容易的。

而软件作为业务的虚拟化，是用来替代业务人员的，业务本身就有其不确定性。在一个企业完成的业务，同行业的另一个企业也不一定能够直接借鉴。并且业务本身就是人的需求，人的需求随社会发展变化的影响太大，也导致无法和建筑一样做到标准化。但是软件相关产业的一些稳定的部分开始慢慢地形成了独立行业，比如计算机硬件设备的云服务就是如此。还有一些软件自身的服务，如监控、报警等，都是一些通用的服务，也可以慢慢地外包或者出租出去了。

随着时间地推进，计算机本身的行业成熟度会逐渐上升，计算机本身的一些业务也会慢慢标准化，会逐渐形成建筑类似的外包或者出租的局面。甚至有可能所有的行业都会变成软件行业的一个分支，完全被虚拟化。行业人员就是软件工程师，软件开发成为人人都必须具备的技能。

有了建筑之后，人的一辈子大部分时间都在建筑里面度过，可以让人的生命周期活动更加舒适、安全，帮助延长人的生命周期时间跨度。而有了软件之后，人的一辈子能够作出更多的创造，达成更多的目标，也相当于延长了人的生命。

27.2 软件和建筑的架构扩展之异同

建筑架构和软件架构都要面对扩展的问题。随着访问人数的增加，访问量对

建筑和软件都是一个大问题。但是二者对扩展的处理是很不一样的。

建筑在设计之初就已经确定了其用途，访问人数的上限就已经确定了，并且受限于建筑物理空间的大小，以及访问建筑的通道和外部路径的空间大小。大型的建筑，比如一个体育场，上限也就是十万人左右，此时进出口都要特别设计，用以防止踩踏事故的发生。在一些宗教场所，有时会涌入上百万人的人群，此时就非常容易产生问题，经常发生大规模的踩踏事件。历史上最严重的踩踏事件发生在 1990 年的麦加，1426 名朝觐者被踩死或窒息而死。如果需要容纳更多的人，则一般需要新建一个建筑来扩展。也就是说，建筑扩展主要是横向扩展。横向扩展的结果，由一个一个的个体建筑，形成了社区、城市和国家。

房价上涨时，一个家庭往往没有足够的资金购买更多的房子来改善居住空间。为了改善家庭建筑空间狭小的状况，也有用纵向扩展的方法来扩展空间的，毕竟装修一下成本更低。其原理在于人在使用空间时，空间的占用和人本身的生命周期活动是有关的。比如人晚上睡觉时，客厅、饭厅等都是闲置的；而在白天不睡觉时，卧室是闲置的。因此，可以通过一定空间隔离措施，使得相同的空间在不同的时间变成不同的用途，在有限的空间内达到比较高的空间利用率，也相当于扩展了空间。

前面也提到，建筑架构师们总希望自己的建筑能屹立千年，不希望自己的设计被重新推翻重来。因此建筑的建设周期以年为单位，是一个漫长的周期，推翻重来的成本非常高。所以建筑架构相对是非常稳定的。

在软件研发的早期，软件的建设和架构是向建筑学习的。软件架构师们希望能够一次性地把软件做好，像建筑一样使用很长的时间，这就是瀑布式的开发模型。但是这个做法忽略了软件和建筑的不同，结果并不如人意。

一方面是由于人们对软件的认识还远远达不到对建筑的认识程度。比如业务人员无法想象软件运行起来是怎样的，可是每个人都见过建筑，建筑模型也很容易展示、体验。另一方面建筑的访问增长是可以预测的，而软件则不可测。比如一个家庭的成员的增长，一般是二十年左右一代，增长率是可以推算的。同理，一个社区的增长也是有限的，增长率也是同样可以推算的。当人数超过了建筑的容量，人们会自觉的建立新的建筑来容纳新增人口。比如下一代往往会去建设自己的房子，城市涌入大量人口的时候，也会建设新的房屋来容纳。建筑的访问增

长是可控的，原有房屋不太会因为访问量的增长而重建。

软件则是不同的，软件虽然在设计之初都有预估访问量，但是访问量总是超过建设者的预期。人们也总是希望用户的访问量越大越好，并不希望去控制用户的访问量。所以软件的用户访问容量上限就是整个地球的总人口产生的访问量。如果软件和计算机技术进步到即便支撑全人类的访问，成本也可以做到很低程度的话，那么也可以采用建筑的方式，做出一步到位的架构设计。可是在当前的技术条件下要支撑这么大量的访问，就需要巨大的成本，同时技术上还有很多的未知数，所以不可能一开始就这么做。因此软件只能慢慢随着时间地推进，不断地重新变更发布，不断地架构拆分增加更多的计算机，以支撑越来越多的访问。这就要求软件的迭代要快，迭代周期往往以周为单位。所以软件建造的自动化程度要远远的超过建筑。

软件的每次发布，就相当于建筑的重新再建造一次。软件的优势是建造非常简单的代码，不仅原有的设施可以直接使用，而且每次发布都可以重用以前写好的代码，只需要做增量的修改即可。而拆掉建筑的成本是非常高的，如果以不损坏的方式拆除，并不比建造的成本低，并且这种拆除的速度很慢，和建造的时间差不多。为了节省建筑拆除的成本并提升拆除的速度，人们往往通过爆破的方式拆除。建筑以这种方式被拆掉之后，原有的建筑几乎不可能被重新使用。建筑的模块化程度也远低于软件，毕竟物理层面的模块化相当的困难，要超越空间和重力的限制。随着技术的进步，现在建筑业也有这方面的发展趋势。

所以，建筑的架构确实是值得我们借鉴的。但是我们不能盲目地借鉴，要针对问题进行分析，再提取对我们有益的营养。表面的类似往往会误导我们。从架构上来说，软件架构和建筑架构本质上都是一致的，都是对生命周期的拆分来达到并行，提升对访问的支撑，提升生产力，提升人类的产出，让人类获得更大利益。但是软件和建筑属于不同的业务领域，各有其自身的业务规律。架构必须要在遵从业务生命周期规律的前提下进行架构拆分。

第三部分

软件架构的应用

以企业的交易业务为例，探讨在企业软件中的业务架构和软件架构，以及应该如何思考和应用业务架构和软件架构

第 28 章　交易

在前文中曾经提到，人类社会的架构拆分形成了分工，提高了每个人的产出，进而提高了整个社会的效率。而在分工之后，每个人的产出都不足以满足自己日常生活的需要。要获得所需的其他生活必需品，就必须要通过交易才行。那么什么是交易呢？

28.1　什么是交易

分工开始后，人们都只做自己擅长的工作，因此产出是非常高的，所产出的物品一定超出了自己的需要。超出自己需要的这部分，则会拿出去用来和其他人做交换，用来获得自己所缺乏的生活必须品。比如种粮食的人，会用自己吃不完的粮食去交换种棉花的人所收获的棉花，用棉花做成衣物给自己保暖；种棉花的人则通过这个交换获得了自己所缺乏的粮食。种粮食的人和种棉花的人通过交换都获得了吃和穿的生活必需品，而各自所需付出的劳动变得少多了，因为都只需要专心做自己擅长的事。这个交换是对双方都有利的，用现代的语言来说，是双赢（Win-Win）的状况。这就是所谓的交易：

交易就是人们各自拿自己多余的物品，从其他人手上换取自己必须的物品，从而双方都获得利益的过程。

所以交易也是有一个生命周期的，从一方开始发出交易的邀约开始，到最后双方物品交换成功结束。通过分析交易，可以发现交易有如下的特征：

- 交易发生在人与人之间，解决的是人的问题。
- 由于不同的人分别处于不同的空间，交易受空间的限制。交易的物品需要从一个地点转移到另一个地点，并转换所有者。
- 交易的发展受时间的限制，交易的进程严格按照时间的推进，直到双方都获得对方的物品为止。

交易实际上是人访问物品生命周期的拆分。分工之前，每个人拥有的物品是自行制造或者从大自然获得的，这时直接访问即可。分工之后，人访问物品的生命周期发生了架构拆分，拆分出了交易生命周期，用来获得别人产出的物品，这时物品先要被获取到才能访问。

交易最早的方式是物物交易，交易发生的地点逐渐集中就形成了市场。所谓的市场，就是指集中发生交易的地方，每个人都可以在市场上卖自己的物品，也可以在市场上寻找自己想买的物品。由于空间的限制，市场往往出现在交通发达，距离人们居住点都比较近的地方。村落内部有自己的市场，村落之间也有集市。当村落的产出不满足自己的要求时，可以去村落之间的集市寻找，甚至去更大的集市。

28.2　货币的出现

物物交易有很多难以克服的弊病。比如在交通不发达的时代，当物体比较大、比较重时很难运输到市场上去做物物交易，因为有空间地域的限制。在天气很热的时候，物品，特别是食品很容易发生腐败，因为有时间的限制。食物运送到市场时，往往费时比较久，腐败了就没有交易价值了。如果物品费时费力运送到市场，但没有人买，还得运回来。这些都是问题。

是不是可以不用把东西带到市场上就可以进行交易呢？为了解决这个问题，人们用一个标志物来代替实际的物品。比如用一个贝壳来代替一千斤的粮食，成交之后买家用贝壳做信物，来获取这一千斤的粮食。这就是信用交易。古代的人非常淳朴，不会去欺骗人，这也是信用交易能得以出现的原因。

刚开始的信用主要局限于买卖的双方，由给出信用物的一方来保障信用的兑现。随着欺骗状况的发生，出于信用保障的需要，信用机构逐渐自发地出现，政府背书的信用也逐渐地形成。随着不守信用的现象出现，信用物本身也在升级，从贝壳等常见物，变成了黄金、白银等贵重物品。这些贵重物品本身就比较难得，并且其生命周期性质稳定，不易随时间变化而发生损坏，体积也很小。因此黄金、白银等通用的信用物逐渐的形成，可以用来购买各种物品，保障信用的兑现。这类通用的信用物，也被称为货币。为了制作和携带的方便，纸币也逐渐地出现。

当信用机制出现后，物物交易的买和卖又发生了拆分，变成了卖家卖出自己

多余的物品，换成信用物；买家通过付出信用物，获得所需要的物品，这就是人们所说的“一手交钱，一手交货”。当卖家自己需要别的物品时，再用信用物来获取所需。物物交易就拆分成了：卖物→信用物；信用物→买物，一次交易变成了两次，买家和卖家都获得了时间和空间上的自由度。交易的生命周期被拆分了，每个个体的买卖同时发生物物交易，在时间和空间上发生了架构的拆分，买和卖分别被执行。人们因此可以更自由地通过交易获得自己所需的物品，交易变得更加容易、方便，市场也变得更加繁荣。

28.3 企业的实质

交易背后的实质，实际上交换的是买方和卖方各自的时间。分工的发生，导致人们从事自己最擅长的事情，也就意味着单位时间内可以得到更大的产出。用单位时间增加的产出去换取另一个人对应时间增加的产出，大家都用比较少的时间得到了更多的回报。因此，慢慢也出现了很多人直接用自己的时间与人交易，得到自己的生活必须品，这就形成了雇佣关系，从而变成了职业。

职业状态下的人们不再为自己干活，而是把自己的时间出售，为其他人工作，用自己的劳动赚取工资，即信用货币。再用获得的货币来购买自己的生活必须品，支撑自己以及家庭成员的生命周期活动的推进。

而雇佣人们为自己干活的人，可以在单位时间内获得更大的产出，在交易时也可以获得更有利的地位，因此也有更大的竞争力和生存能力。随着人类社会的架构拆分，雇佣人的方式也在慢慢地发生变化，逐渐从家族的形式变成了现代的企业。企业通过对生产的生命周期分工，雇佣更多的人，组织他们并行工作，可以得到更大的产能，甚至达到 1 + 1>2 的效果，这就是企业的组织架构。个体的产出因此汇集成了一个集体的产出，这个集体成了一个更大型的“人”，可以做到个人无法达成的产出。企业的出现使得人类的生产效率得到了急剧地提升。企业从人类社会中拆分了出来，形成了一个新的生命，也有其自身的生命周期。

因为交易的出现，使得这个世界的人们能够更加紧密地团结合作，每个人的抗风险能力都得到提升，大家都得到了双赢的结果，形成了人类社会，一个新的生命出现了。

28.4　软件对交易的影响

软件和互联网发明出来后，交易也开始用软件进行虚拟化。人们在互联网上售卖物品，或者购买物品。于是市场也开始虚拟化，形成了一个虚拟化的集市，人们可以足不出户地在网上浏览、搜索自己需要的物品，从而突破了空间的限制。售卖的人也不再需要占用一个物理空间作为实体的店面，因为网络上就可以虚拟出一个商店，供人浏览、购买。

交易虚拟化之后，由于人本身的生命周期活动还无法虚拟化，所以购买的物品在物理上还是要被自己使用，交易的生命周期被进一步拉长，买卖双方在达成交易之后，卖家再通过物流把物品传输到买家。

由于物流的出现，买家付款后无法及时的收到货物，所以“一手交钱，一手交货”的规则不再适用。而卖家没有收到款项，也无法把货物发给买家。也就是说，交易进一步拆分之后，由于不再是空间上面对面的交易，买家和卖家都无法充分的信任对方。为解决这个问题，交易生命周期被进一步的拆分，第三方支付担保（Escrow）出现了。第三方支付机构是一个对买卖双方中立的机构，具备一定的信用，类似于银行。买家把钱付给第三方支付担保公司，由第三方支付担保公司通知卖方发货；卖方的信任问题解除，发货给买家；买家收到货之后，通知第三方支付担保公司把钱真正支付给卖方，买卖双方的信任问题解决了。

交易虚拟化之后，货币的传输还是通过汇款的传统方式，这种方式效率很低，也引发了虚拟化，即所谓的数字货币。

由此可见，交易的软件虚拟化导致交易发生了新的架构拆分，形成了新的社会分工，交易相关的要素都完成了虚拟化。交易虚拟化后，人们完成交易的难度进一步降低，极大地节省了交易参与者的时间和空间成本，人类社会的效率得到进一步的提升。

28.5　软件的交易

人们需要交易，实际上就是为了获得物品，独占对物品访问的权力。软件出现后，为了获得真正的物品，在访问到真正的物品之前，先要访问到软件来获得对软件访问的权力，软件也变成了一个可以获得的物品。也就是说，用户和软件之间也发生了交易。促使用户访问软件的动力，就在于软件的虚拟化极大地减少

了人们获得物品的难度，所能获取的信息量更大。

因此要评价一个软件的价值，它的访问量是一个至关重要的衡量标准，因为每一次用户的访问，就相当于该软件所产生的一个交易。用户通过付出访问时间以及个人的信息，来获得软件所提供的服务。软件提供方则通过大量的用户访问，积累了巨大的用户访问流量。通过对用户的访问和个人信息的分析，可以让这个软件很容易地根据用户的习惯，主动把用户需要的信息提供给用户。市场营销也变得非常有效，而且减少了向用户营销的空间和时间上的限制。因此企业的运作模式也发生了巨大变化，一批企业不再直接和用户发生金钱的交易，而是通过流量的分析以广告等方式进行变现。也就是说，用户对软件的访问已经是一个交易了，软件提供商和用户互相都通过访问得到了自己需要的东西。同时，这类企业的出现也导致用户对访问目标的访问路径出现了一个新的架构拆分。

对于一个提供搜索的企业，用户的每一次搜索都是一个交易。对于提供网络营销的企业，用户对广告的每一次观看、点击都是一个交易。所以，为什么会有PPC（Pay Per Click）、CPC（Cost Per Click）等付费方式就可以很容易理解了。这些都属于交易。这就是为什么前文一再强调，访问生命周期是如何的重要，一次访问就是访问者和被访问者两方的一次交易。

很多人会有疑问，没有发生货币的转移，怎么能算交易呢？

交易的背后实际上是人们互相之间时间的交换，而不是货币。货币只是对时间的定价，方便交易的一个工具。

对于不同的商业模式，交易都有自己的形态，需要人们识别出来。

识别的方法，就是看在哪里发生了时间的交换。

28.6 企业的核心

从企业的形成历史来看，企业是一个由很多不同的人组成的一个新的生命，有自己独特的生命周期。人的生命是有限的，而企业的生命长度可以远远超过一个人的生命长度，所以很多人会把企业当作自己生命的延长来付出。企业要具有比人长的寿命，就要超出个人的诉求，不再为某个个人服务，而是为了一个群体服务，形成了企业的核心诉求和目标，也就是人们常说的企业愿景（Vision）。这

个愿景决定了企业生命的长度。并且这个愿景也不是固定不变的，往往会根据社会变化的进程，根据企业领导人的思考进行调整，随着社会的变化而变化，以求能够生存得更久。一般来说为整个人类进步而思考，紧跟人类社会变化而变化的企业，往往具备比较长的生命。为什么是这样，留给大家去思考。

企业在达成愿景的过程中，其生命周期由一个一个和用户的交易积累而成。一个一个的交易，又由一次一次的访问生命周期积累而成。这就是企业生命周期的运作过程，也是企业生命周期的架构拆分。由于这个拆分，形成了企业的组织架构。

组织架构的目的就是通过更多的人的并行工作，来支撑更多用户的访问生命周期。为了增加企业和目标用户的交易，需要增加企业的访问量，需要扩大用户群，还需要拆分出很多其他的生命周期为交易生命周期服务。因此，一个企业的整个架构体系，就是围绕着以交易为核心的一个树状结构。这个架构体系的目的，也是为了让交易过程中所有的参与者，不仅能够获得自己的利益，而且能用更少的时间获得更多的产出。只有树状架构才能保证权责的一致，确保双赢的达成，使得每个人都能够从交易中获得激励。

对于不同的企业，什么是交易，取决于这个企业的愿景是什么。对于售卖型的企业，其交易就是每一个物品的售卖，如产品的销售等。对于通道型的企业，其交易就是通道的每一次访问，如代理商、超市、在线购物平台等。对于营销类的企业，用户的每一次关注，也就是每一次访问，也是一个交易。无论哪种企业，交易最终的实现，都是通过用户的访问来达成的。访问是如何达成的，访问生命周期是如何切分的，则是不同企业的区别之处，也是不同企业愿景的入手点。毕竟交易要通过访问才能够达成。

第 29 章　产品

交易的过程中，人们交换的物品被称为产品（Product），有时也被称为商品（Good）。那么产品和商品究竟有什么区别呢？还是两者同一个意思？

29.1　什么是产品

产品这个概念，也是一个典型的缺乏主语的例子。产品配上主语就是：谁生产的一个东西。所以产品想表达的实际上是某个主体所生产出的东西。比如石头、山、林、树木等是大自然的产品，鞋子、计算机、桥梁等是人类自己生产出的东西，是人类的产品。再精确一点，苹果电脑是苹果公司生产出的产品，长白山人参是长白山生长出来的产品等。

任何事物，一定都不能自己产生自己，一定有其产生的主体，这个主体一定不是这个事物自身，这是产品的另一层含义。如果违背了这一点，事物就永远不会消亡了，生灭法则就破坏了。生灭法则破坏了，这个世界就停止不动了，也就没有生命周期了。

从人类的视角出发，产品生产的主体可以分为人类和自然界两类。每个产品本身的名字或品牌往往就已经指出了这个产品的生产者是谁，有可能是人类，也有可能是自然界。

产品本身的名字也往往和该产品所解决的人类问题有关，比如鞋子是用来保护人类脚的，桥梁是为人类连接自然界的江河所割裂的地面的，等等。所有用来交易的产品都是为人类的日常生命周期活动服务的物品。

人类要使用产品，即访问产品，只有有限的几个方式，且和人体的结构有关系，古人已经总结的非常到位：“眼、耳、鼻、舌、身”。这几个方式主要感知这几类事物：“色、声、香、味、触”，产生相对应的几种感觉如下：

“色”不是“情色”的意思，而是指眼能够看到的景色，比如“青、黄、赤、白”，“长、短、方、圆”等，对应的是视觉；

“声”为耳朵所能够听到的声音，包括一切言语音声等，对应的是听觉；

“香”为鼻子所能够闻到的气味，好闻的、不好闻的等，对应的是嗅觉；

“味”为舌所能够尝到的味道，如“甘、酸、苦、辛”等，对应的是味觉；

“触”为身体能够感受到的感觉，如“冷、热、软、硬、涩、滑”等，对应的是触觉。

人通过这些感觉器官形成了对外部世界的访问，包括对人体本身的访问。软件出现之前就已经出现了对人类世界的模拟，也就是影像类的产品，比如照片、电视、电影等。早期的照片和无声电影，通过模拟视觉所看到的景象来虚拟化这个世界。后期的有声电影，通过同时模拟视觉和听觉所感知的信息来虚拟化这个世界，让人更加地沉浸于其中，随着电影中主人公的命运，情绪也跟着波澜起伏。不过嗅觉、味觉和触觉的模拟，至今还没有太多的突破，未来应该会有进展。如果这三样都得到了突破，人类是不是躺着不动就可以享受整个人生了呢？那和做梦有什么区别呢？人生的意义就值得思考了。这里就不展开探讨了，问题留给大家。

既然人类对产品的访问都是通过这几个感觉来进行的，而这些感觉又都是需要通过空间才能够访问的。因此产品的生产方就必须要以某种方式来突破空间的限制，使得用户能够在空间上到达产品，才能够帮助用户解决问题。这个方式就包括了物理的方式和虚拟的方式。物理的方式很好理解，虚拟的方式就是在前面所述的影像的模拟和声音的模拟，如电子书、音乐等就是这种方式。而交易的目的就是为了能够让人类在访问产品前先获得产品。

所以人类要访问产品，首先要有一个访问目标，即产品本身，其次还需要一个访问的途径，两者都具备才能够达成对产品的访问。由此可以认为人类生产的产品主要可以分两大类：制造类的产品和服务类的产品，服务类也可以称为通道类。

制造类产品是人类访问的目的地，通过产品本身的生命周期变化帮助人类解决问题。这类产品用自身的生命周期变化的推进，支撑了人类的生命周期活动。比如鞋子通过自己的磨损，来保护人类的脚。鞋子磨破了不能穿了，鞋子的生命

周期就结束了；食物通过自己生命周期的结束，给人体带来能量；电影通过一帧一帧图片的消失，形成连续的图像，供人欣赏，直到电影结束；等等。

制造类的产品要能够让人类访问，必须先要到达人类的感觉范围内。因此必须要建立空间的渠道，经历时间传输到用户的感觉范围内，用户通过自己的感觉器官才能够接触访问到产品。根据制造类产品的类别，还有两种不同的情况：

- 物理类的产品往往要到达用户的触觉，再形成专属的访问。比如鞋子要从生产厂家传递到用户脚上，才能够真正形成对用户的脚的保护，可能要经历物流、超市等渠道类产品才能够达成。
- 虚拟类的产品则不一定要到达用户的触觉，可以在用户访问的时候再形成专有的一个空间访问通道，对用户的访问生命周期进行支撑。当然访问这类产品的前提是拥有一个可以访问的终端，这一般也是一个物理制造类的产品。比如在线内容提供商提供的就是这样的产品，如电子书、电影等，可以在线阅读，在线观看电影，但是要一个终端来播放。当然虚拟类的也可以做成物理类的方式传输到用户处，比如通过 CD、DVD 等进行传输，在用户这里通过物理设备完成虚拟产品的模拟，并形成访问。

为人类提供对制造类产品访问的空间通道，所形成的便是服务类的产品，也就是通道类的产品。服务类的产品通过访问通道，解决人类对外部世界的访问生命周期问题，连接人类和访问的目的地。

比如“农夫山泉矿泉水”，这是一个产品，表达的是农夫山泉这个公司生产出的一个品牌的矿泉水。水是人类生命周期活动中至关重要的东西，必须通过触觉，让水进入人体，人体才能够访问水。而水是大自然产生的，属于大自然的产品。“我们不生产水，我们只是大自然的搬运工”，这句广告语并非没有道理。既然水不是农夫山泉这个企业的产品，那么农夫山泉这个企业的产品究竟是什么呢？其实这个企业的产品是用户喝到水的一个通道，也就是消费者能够喝到大自然的水的一个访问通道。这个通道把水从大自然引到消费者手中，让消费者可以访问到大自然的水，是对人访问自然水的访问生命周期的一个切分。

同样，一个超市售卖这个矿泉水，超市的产品是什么呢？超市的产品就是超市本身的这个售卖渠道。超市通过提供一个独占空间，方便人们获得所需要的物品，这就是超市的产品。

那么农、林、牧、渔等算是什么样的产品呢？它们是自然界的产品。人类借助自然的力量对环境进行控制，根据各种生命自身的生命周期规律来人工介入养殖，或直接获取，所形成的也是人类访问自然界的一个通道，属于服务类。事实上人类的任何产品都是来源于大自然的，都是地球带给我们的，人类所做的只不过是利用大自然的规律，把大自然的产品转化为人类日常生活所需要的东西，美其名曰人类的创造。

人类绝不能以牺牲地球的代价无休止地向地球获取，长此以往地球的自我保护机制启动，地球将不再适合人类的生存。地球的能量其实也来源于太阳，太阳能是非常清洁的能源。当人类突破太阳能的获取方式，得到清洁地获取太阳能的技术，如树木等生物获取太阳能的方式，人类将可以不通过地球为中介，不再以破坏地球环境为代价来获取大自然的资源，形成健康的资源获取生命周期循环，与地球以及地球上其他生命和谐地生存在一起。

还有一类制造是无法送到人类身边的，也无法突破空间的限制，并且暂时也无法做到虚拟化。比如大自然所创造的自然景观，人类社会所累积的人文环境等。既然这些产品无法送到人身边，那么就把人送过去，毕竟人是可以穿越空间的。因此就出现了很多旅游企业。旅游企业的产品，也是通过形成一个独占的访问通道，帮助人类能够体验到制造类的产品。如旅行社通过组织汽车、火车、飞机、门票等形成旅游线路，酒店提供住宿等，这类也是服务类的产品，形成的是人类的访问通道，给人一个独占的访问空间。人们手持的汽车票、火车票、飞机票、门票、酒店的房号等，都是在某段时间内可以独占一个空间的凭证。

这两类产品的质量评价标准也有显著的不同。制造类的产品，以在其产品生命周期内对人类生命周期活动支撑的有效性来评判，人类对其访问生命周期支撑的越多，访问生命周期的结果越好、质量越好。比如鞋子，鞋子穿得越舒服，支撑人类脚部活动的时间越长，则质量越好。通道类的产品，往往以访问的流量、便利性取胜，也就是让每个访问生命周期的时间越短越好，同时能支撑的访问量越大越好。比如超市往往开在人流量大，离人近的地方。访问通道对人的方便程度，就是这类产品的质量。

不管是哪一类产品，目的都是为了解决人类本身的生命周期活动问题。人类要解决自己的问题，需要通过组合访问这些产品来达成。

29.2 什么是商品

当售卖产品的时候，还能不能用产品这个概念呢？比如用户买十瓶农夫山泉矿泉水，用十个产品来表达似乎不太对，哪里不对好像又说不太清楚。

产品和商品的概念还是有区别的，因为两者所解决的问题不一样。产品更多的是表达生产属性，而商品则更多的是表达交易属性。产品本身不一定能够拿来售卖，比如有些自娱自乐的产品，或者一些国家禁止的产品等。但是售卖的东西一定是某个产品，也就是说一定是某个主体所生产的产品。

比如对于一个鞋子厂商，售卖的是自己生产的鞋子，鞋子直接到消费者那里，访问通道很短。前面已经知道，要做好交易必须要建立和客户的访问通道。鞋子厂商的长处在于制造鞋子，在建立客户访问通道这方面显然并非所长，自然而然会有擅长售卖的企业介入，来帮助鞋子厂商售卖产品，比如一个超市就是专门售卖的。擅长建立通道的企业，所生产的产品就是和用户所建立的通道。通道企业一旦介入，用户得到鞋子的访问生命周期就发生了拆分，形成了新的通道生命周期，这个新的通道生命周期形成了新的社会分工，也就是专门做通道的企业。

通道企业建立之后就不会只卖鞋子，因为通道变成了他的产品。因此该企业就要增加访问量，卖更多的产品，卖的产品越多，通道的成本就越低。因此，不管是制造类的企业，还是通道类的企业，都是在卖自己的产品。区别在于通道类的企业是借助制造类的产品来完成自己产品的售卖。用商业模式来说，通道类企业采用的是 B2B（Business to Business）、B2C（Business to Customer）等模式。

而从交易的角度来说，交易并不关心产品本身所解决的问题。产品的理念往往都是很好的，但是是否被人们接受，就要靠交易的达成。而交易时只关心产品的数量和价格，价格往往反映了市场的供需水平，不是产品这个概念能够表达的。所以此时往往采用商品这个概念，用来表达产品交易的属性。比如超市售卖的一瓶农夫山泉矿泉水就包含了两个产品：一个是农夫山泉矿泉水本身这个产品，另一个是超市渠道本身这个产品。而超市售卖的商品对用户来说只有一个，即一瓶农夫山泉矿泉水，超市自己渠道产品的价值实现就包含在所售卖的商品价格中。

所以什么是商品呢？用来交易的产品单元才是商品。比如一瓶农夫山泉 500ml 矿泉水和一箱农夫山泉 500ml 矿泉水是两种商品，可它们都是同一类产品。

从用户角度看，在达成交易之前，用户寻找的是解决自己问题的产品。而在寻找到自己想要的产品之后交易时，用户考虑的则是数量和价格，交易时购买的是商品。在用户使用的时候仍然是产品，比如会说自己喝的是农夫山泉矿泉水，也就是产品属性。

从售卖产品的商户角度来看，使用的虽然是商品概念，但是离不开产品的概念。比如要依照产品本身的特性、产品本身所解决的问题等，分门别类整理好商品，方便用户容易查找到所需要的产品，为进入交易生命周期做准备。库存没有的商品，就要及时下架。而为了管理库存，每种商品的存量要按照用户的需要来配置，存货的单位就是商品，往往也称为 SKU（Stock Keeping Unit）。就库存来说，库存并不关心产品的含义，只关心商品的数量、重量和体积等空间因素。因为库存就是交易物品的存放，是为交易准备的，并且需要占用空间。

因此，商品是为了完成产品的交易而形成的一个概念，它和产品息息相关，离不开产品，又独立于产品。

29.3　识别产品

对于一个企业来说，如果这个企业的领导人对自己企业的愿景不够清晰的话，往往会带来一个严重的问题，就是这个企业的产品往往是模糊的。产品不清晰导致的结果就是：目标人群不清楚，产品体系混乱，进而导致组织架构混乱，交易很难增长，企业也会陷入困境。就好比一个人一样，如果这个人对自己的生活目标不清楚，不知道自己能够给其他人带来什么价值，这个人就很容易陷入困境，不知道自己能干什么，生活会变得一团糟。

人也不可能一开始就把问题想得很清楚，总是有一个时间推进的过程，发现问题也是有一个生命周期的。发现企业的愿景和产品这个生命周期活动的主要内容就是通过和用户的沟通，通过用户的实际访问生命周期的搜集，慢慢来认识用户的真正问题是什么。这就是为什么会有商业智能（Business Intelligence，BI）、数据仓库等出现。人也总是希望沟通的反馈越及时越好，因此大数据的实时处理基础也出现了。通过这些沟通的反馈，企业的愿景会越来越清晰，产品也会越来越符合用户的需求，企业的生命力也会越来越强。

还有很多企业，莫名奇妙做得很成功，但是不知道为什么。莫名其名的成功

往往会导致莫名其名的失败。此时要及时发现自己成功的原因，认识为人们所解决的问题，以此为契机逐渐形成并提升企业的愿景。

基于前面的讨论可知，企业的产品来源于企业对自己在人类社会的分工定位，也就是企业的愿景，产品的价值来于对人类问题的解决。因此识别企业的产品，可以通过理解人类生命周期活动对外部的访问生命周期的拆分开始，发现企业在人类社会拆分树上的位置，基本上就可以确定企业的产品。

识别出产品后，就可以根据自己产品的特性，针对不同的客户群，形成不同的产品型号。比如有些超市提供 VIP 通道、年卡服务等。

29.4 产品系统

用户要交易，首先就要知道有什么产品可以购买，因此必须要有一个产品的展示入口供用户浏览。而要让用户知道这个产品，就必须要把产品所解决的问题等信息传播出去，使得目标用户能够访问到这些信息，通过这些信息吸引用户来访问产品的入口，让用户产生交易的意愿，这就叫做市场营销（Marketing）。一切人类能够接收的方式都可以采用，比如视觉、听觉、嗅觉、味觉和触觉等方式，还要能够容易到达用户的感觉范围。在没有软件系统的时候，这一工作往往通过纸面的介绍或视频、声音等的传播来进行。

比如古话说“酒香不怕巷子深”，这是因为酒香可以随空气飘得很远，可以覆盖比较广的空间，此时空气成了传播的媒介。而香味属于味觉，在激发人的食欲方面有重要的作用，就激发了闻到香味的人来买酒。这其实就是典型的市场营销，利用了几乎没有成本的自然风，核心是酒这个产品本身的吸引力。现代很多人以为这句话的意思是不做营销，其实是误解了。应该理解为卖酒的老板，利用产品自身的属性，低成本甚至无成本地营销。

产品本身就是最好的营销。

通过酒香的传播，建立了产品访问通道，吸引了好这一口的客户。如果喝过的人觉得很好喝，还会主动给周围的人推荐，口口相传，这也是营销。所以：

产品系统实际上是市场营销的起点，也是交易的起点，也是企业用来实现自己愿景的工具，是愿景的具体化。

29.5　产品列表

要做好产品系统，首先要做的就是产品列表，或者叫产品目录。产品列表要展示的信息，就是把企业的愿景传递给用户，取得用户的认同并购买。把产品列表向目标人群分发，就是产品的营销。

为了把产品介绍给用户，就要把产品带给人们的利益介绍清楚。因此产品的列表，要包含产品本身的能力及其能解决的问题，往往还要给产品定一个名字来代表所解决的问题。比如鞋子，我们都知道是来保护脚的，鞋子厂商的产品列表，一定是针对人对鞋子的不同要求的，比如干活用的、跑步用的、防水用的等；再比如超市，就是方便人们买东西的一个市场，其产品列表一定是所售卖的不同产品的名字。

而为了把产品导向交易，产品列表需要标明商品的价格。价格是用户产生购买意愿的非常重要的参考指标，因为用户通常会“货比三家”。打折和促销信息也会出现在这里，用来提升对用户的吸引力。库存是否足够的信息，也会放入产品列表。比如超市的产品列表，有时会标注“仅剩多少个”等，提醒用户不要错过好机会。

29.6　产品详情

产品列表就相当于是一个索引，帮助用户从众多的产品中找到符合自己需要的产品。但是产品列表中的信息非常的有限。用户要下决心购买，还需要进一步了解产品的详细信息，所以一般会有一个产品详情，从各种角度完整地介绍产品。好的详情信息，会进一步坚定用户产生交易的决心，一般包括如下几点：

产品本身的详细介绍，如产品特性、产品保修和售后规范等。

还会列出已购买用户对产品的评价，增加用户的信任。

产品本身的促销信息等，引导用户交易。

列出这个产品所包括的商品种类，方便用户选择下单。

好的产品详情不仅能增加用户对产品的信任，还能增加购买的动力，并能方便用户进行下一步的下单动作。

29.7 商品的规则

为了提升销量，或者吸引更多的用户，商品售卖时往往会进行优惠促销。因此每种商品往往会有自己的一些售卖规则。这些规则总结起来主要分为以下几类：

从商品本身为出发点的，一般是来提升商品本身的销量。比如商品在某段时间内优惠多少、优惠某个固定的价格、优惠某个比例等。或者是为了提升商品的评价参与度，通过评价后返还优惠的方式，激励用户反馈商品的质量，形成售卖的生命周期良性循环。

从客户本身为出发点的，一般是用来提升用户的数量，或者提升用户的活跃度。比如新用户注册优惠、生日优惠、首次购买优惠等。

从支付推广为出发点的，一般是用来提升某个支付方式的使用率。比如针对某个信用卡的付款优惠，针对现金付款的付款优惠，分期付款的付款优惠等。

如果企业自身的产品是渠道，企业的供应商往往对渠道有商品价格的限制，用来防止不同的供应商恶性竞争，损害供应商自身的利益。而企业为了提升渠道本身的流量，规避供应商的价格限制，企业就通过对商品本身的交易付款时进行前返或者后返，形成渠道本身对用户交易的激励，附带在供应商提供的产品上。

可以看到，这些优惠规则最终都要体现在商品的交易上，通过交易过程本身的行为达成激励的兑现。而交易过程中，商品的传递是贯穿始终的，把这些激励的方式形成规则，通过商品作为载体来传递就自然而然成为了一个最好的选择。

第 30 章　用户

要完成交易，就不能忽视交易的主体，毕竟交易是为了给参与的主体们达成双赢的。那么什么是用户呢？在现实生活中，人们又是如何看待用户的呢？

30.1　什么是用户

用户，顾名思义是和“用”有关系。用的是什么呢？用的是某个主体生产的产品。当某人正在使用某个主体生产的产品时，一般会说此人就是这个产品的用户。所以用户这个词的出现，和分工是有关系的。当一个产品的生命周期拆分为生产生命周期和运行生命周期之后，产品生产的主体和产品使用的主体，即访问的主体，就发生了架构拆分，变得不再一致，此时用户的概念就出现了。

用户表达的是产品运行生命周期中所服务的人或人群。比如某个工厂生产的鞋子，被某人买去。那么鞋子是该工厂生产的产品，而买鞋人是该工厂所生产鞋子的用户。同样，一个超市售卖这个鞋子，某人去超市买这个鞋子，这个超市作为渠道，买鞋人是这个渠道的访问者，也就是这个渠道的用户，该用户购买并使用这个鞋子之后，就成为了这个工厂生产的鞋子的用户。

如果要说用户，一定要说明白是哪个产品的用户，避免产生误解。由此也可以看出，“用户”是一个典型的缺乏主语的概念。

与用户相关的还有一个词汇，叫作客户（Customer）。客户和用户有什么异同呢？客户这个概念更多的是关注在交易层面。当一个人去某工厂购买该工厂生产的鞋子，该工厂作为渠道，买鞋人是该渠道的用户，正在访问、使用工厂的设施。当这个人看到鞋子的信息准备挑选购买时，这个人是该工厂销售人员的客户，销售人员要把该渠道的用户发展成为该工厂所生产产品的用户。也就是说客户是产品销售的对象，也是交易的对象，目标是为了让产品的客户成为产品的用户。

但是用户和客户这两个概念在实际的使用过程中往往被混用，从而进一步加

深了业务理解的混乱和沟通的困难。因此，理解这两个概念背后所解决的问题至关重要。

30.2 为什么需要用户

作为企业，为什么要关注用户这个概念呢？企业的目标是通过生产产品来满足人们生命周期活动的需要的，企业针对的是哪些人群？这些人群的生命周期活动有哪些特征？企业的产品是如何覆盖这些目标生命周期活动的？这些思考的结果，成为了企业的愿景。而产品生产出来之后，产品的运行生命周期是在用户处发生的，访问的结果是保存在用户这里的。人们对企业产品的访问如何，是否达成了企业的愿景，企业是很难知晓并跟踪的。可是企业又需要这些信息，方便理解自己的产品是如何满足用户需要的，或者改进产品，或者开发新的产品，甚至调整自己的愿景。这些信息还能够帮助企业扩大其交易量，提升用户数量，产生更多的价值。

从这里可以看出，用户这个概念是针对产品生产者，也就是企业而言的，目的是为了把用户处的信息，通过某种方式搜集到企业这里，是企业对其产品使用方建档并跟踪的一个原始数据组织。

任何产品都需要用户，因为产品的生产都是为人服务的。用户对于企业是至关重要的，任何企业都是由人推动往前发展的。企业只是人类需求的一个实现形式，因此企业时刻都要以人为本。自然界的产品虽然目的不是为人服务的，但是有了人之后，自然界的产品都成了人可以利用的服务。

在现实生活中，用户随处可见。人们只要发生了访问行为，就已经成为所访问的目标产品的用户了。这是理所当然的，但它是如此的常见，以致于人们总是忽略了。

同样，人在访问产品的过程中，人的行为在时间上是连续发生的。人的访问途经也非常的有限，如视觉、听觉、嗅觉、味觉和触觉等，这些都和空间有关，都有距离的限制，需要经历时间的跨度去体味。人的访问行为的结果，会保留在人的记忆里或者人所拥有的物品中，推动人自身的生命周期向前发展。

30.3　客户的出现

现实生活中，要让人们成为用户，首先要让人们成为客户。而为了让人们成为客户，又必须要让人们成为用户。看起来很晕，这是因为忽略了主语的缘故。把这段文字中缺失的信息补全，也就是说，要让人们成为企业产品的用户，企业必须要先把产品卖给客户。而由于前述人在物理空间访问产品的限制，产品要卖给客户就必须要有一个空间的场所，让客户可以在空间上形成访问通道。这个通道一般都是在人流比较多，空间比较开阔的地方，可以让人们不受限制地访问，通过视觉、听觉触觉等来感知。客户要先成为这个空间通道的用户，这样才有机会让客户变成企业的产品的用户。

这个过程和企业创造产品的顺序恰恰相反。企业先要有产品出现，再建立用户可以访问产品的通道，用户则要先访问通道才能够购买产品。这是因为企业和用户行为的推进恰好是交易中买和卖两个相对的方向。

用户要先访问通道才能购买产品，这就是为什么很多商店的装修要符合企业产品的品位和价值。这个通道是为所售卖的产品服务的，体现的是产品本身的品位和价值，因此这个环境本身就可以用来过滤不同的客户。不喜欢或者不需要的用户自然就不会访问这个通道，更不会访问到产品。喜欢的就是这个企业产品的潜在客户或者老用户，就被吸引过来了。

把人们吸引过来之后，这些人就是销售的对象，也就是客户。销售人员的目标就是要把客户变成产品的用户。因此销售人员要和客户沟通，搜集客户的需求，并和产品进行匹配，来推进交易的发生。没能产生交易的客户，也要搜集数据，用以反馈企业产品的战略方向和实施情况。

30.4　用户的生命周期

一旦人们被吸引到产品的访问通道上成为访问通道的用户，人们就成了企业销售人员的潜在客户，也就成为了企业产品的潜在用户，用户的生命周期就产生了。企业需要把这个用户记录下来，作为用户活动信息搜集的起始，来运营并维系用户。这个用户在使用这个产品过程中的信息，都要记录到该用户的名下，包括交易本身的信息。哪怕交易没有成功也要记录下来，因为代表用户的生命周期活动，可以作为下一次销售的基础。

用户使用产品过程中产生的信息，企业是没法知道的，这些信息都保存在用户处。如果在产品上放置自动回传用户访问信息的装置是否可以呢？这又涉及到用户的个人隐私，毕竟产品的所有权已经为用户所有，产品产生的数据当然也是属于用户的，不再属于企业。因此，用户的产品使用行为记录基本上就靠用户主动反馈的产品使用信息，比如产品的异常、质量、保修等问题。企业有责任帮助用户解决产品访问和使用上的任何问题，这就是产品的运营。如何让用户愿意反馈产品的访问信息呢？往往在产品交易成功后，提供现金返还或券的方式来激励用户回馈信息，也称为后返，相当于变相购买用户的访问信息。由此可见，用户对产品的使用，即访问生命周期的信息，是极有价值的。

在用户所使用产品的生命周期结束后，用户的生命周期也就结束了。但是该用户的信息一直保留着企业处，这是企业非常宝贵的数据，也是下一轮销售生命周期的起点。

30.5 用户的识别

现实生活中，用户的概念好像是挺自然的事情，以致于人们往往忽略了人是如何识别出用户的。因此有必要把这些隐藏的信息提取出来，进一步的进行分析。

识别用户，还是要从人本身的特殊结构说起。比如在现实生活中的某个商店，商店的用户在哪，如何判断呢？这里就可以看出人类本身的强项了：人类本身的识别能力，其中最主要的就是视觉和听觉。从人们进店开始，人眼就开始识别了，店主通过人眼地沟通，从访问的人群中识别自己的客户。哪个是老用户，哪个是新客户等，此时店主已经开始在自己的心中建立起用户的档案了。再比如支付刷卡，其实就是身份验证，刷卡留下了纸面的记录。即使用现金购买，购买完成后手持的保修凭证、收据或者发票都是身份凭证，也帮助店主留下纸面记录。

但是在现实生活中，这些用户资料往往散落在各处，大家都没有意识到这些数据的重要性。在用户访问的过程中，这些资料反映了两方面的信息，一方面是通道本身作为产品的访问情况，另一方面是通道本身所售卖产品的访问情况。一般店主都会关心自己产品的访问情况，也就是通道的情况，把这部分用户的信息搜集起来，帮助自己提升通道的流量。而企业的产品访问情况，则往往被忽略了，因为没有形成权责的对等。所以企业应该投入资源从通道处获取自己产品的访问情况，用来提升自己的产品。

第31章　订单

随着人类社会的发展，交易的生命周期逐渐发生了拆分，形成了更多的生命周期活动。首先，交易在空间上发生了拆分。交易发生的地点和交易结束的地点变得可以不一致；其次，交易发生的时间和交易结束的时间也不再连续，被拉长到了更长的时间，且中间也允许被打断，人们不用等交易结束就可以去做其他事情。也就是说，最初交易“一手交钱一手交货”的原则发生了变化。

由于交易在空间和时间上发生了拆分，对企业来说，要和很多客户同时进行交易，如何把这些进行中的不同交易跟踪并记录下来，又不把这些不同客户的交易搞混，并在合适的时候继续交易的进程，就成了一个问题。因此，每个交易的生命周期的跟踪就非常有必要了。

31.1　什么是订单

为了把每个交易的生命周期过程串联并记录下来，并和客户形成一对一的交易跟踪，就形成了订单的概念。所谓订单，就是指和客户的一次交易的生命周期过程。订单也是一个缺乏主语的概念，订单指的实际上是所售卖产品的订单，表述的是产品的售卖过程，售卖的东西是产品中的某个商品。订单所代表的就是一次交易的整个生命周期过程，从交易生成开始产生订单，到交易结束为止完成订单。

社会分工发生后，企业或者个人为了卖出自己的产品或时间，达成交易，核心就在于获得订单。因此，一个企业的核心就是为了得到订单。一个人为了获得自己的工作，也是为了获得订单，用来支撑自己的核心生命周期运转。所以从企业的角度，任何企业的核心生命周期都是为了获取订单，通过源源不断的订单来达成企业愿景。这就是订单在企业中的价值。

就好比人一样，人必须要呼吸、吃饭、喝水才能够维持人的生命周期的进程。

企业则是通过生产产品，不断获得订单才能够维持企业生命周期的进程。职业化的人也是一样，通过获得订单，变成货币来购买自己的生活必需品，用以维持自己的核心生命周期运转，如吃饭等。

产品不同，订单的形式也会不同，因此识别订单就是一个非常重要的事情。比如对于一个软件，每次用户的访问都是一个订单，因此软件的访问日志表示的就是软件自身的一个订单，可见访问日志对于软件是多么的重要。对于一个做搜索的网站，用户的每次搜索都是一个订单。对于一个做广告营销的网站，用户的每次浏览或者点击都是一个订单，PPC（Pay Per Click）、CPC（Cost Per Click）等就是基于和用户的订单来收费的。有些订单比较复杂，有些订单比较简单。简单的交易是最直接的物物交换，因为发生货币的来往所以容易被人们忽略。人们忘掉了最原始的订单就是物物交换，只是这种订单的生命周期很短而已。

人们往往都忽视简单的东西，而把注意力都专注到复杂的事情上了。

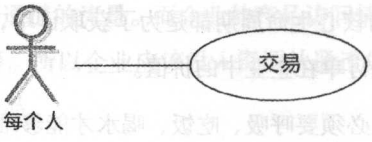
其实复杂的事情一开始都是简单的，简单的东西逐渐地发生拆分就变得越来越复杂了。

就如一棵树一开始只是一颗种子，种子不断地发生拆分长大，就形成了一棵复杂的树，拥有了漂亮繁复的枝叶。明白了这一点，就不再会被漂亮的枝叶所迷惑，不再会被复杂的表面现象所迷惑了。人类订单的发展就是如此。

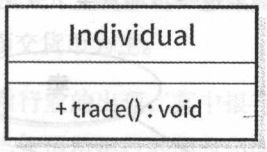
31.2 订单的生命周期架构拆分

在人类还没有发生分工的时代，是不需要交易的。每个人自行从自然界获得生活必须品自力更生，或者也可以认为人类和大自然发生了交易，人类一直从大自然获取而回报甚少。

在分工发生后的物物交易时代，人们以物易物进行交易。人们在完成商品的交换时，交易瞬时达成，在空间和时间上都是连续的。此时订单的生命周期很短，每个个体同时在发生交易的行为。

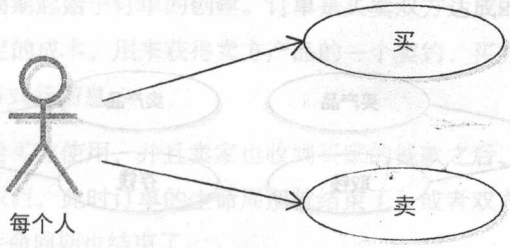


因此在发生分工后，每个个体作为一个生命周期主体，增加了交易这个生命周期动作。

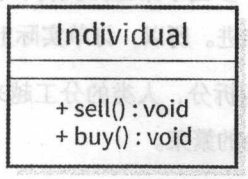


随着信用货币的出现，交易的生命周期被拆分为卖和买这两种行为。每个个体卖和买的行为在时间及空间上同时发生了拆分。在时间上，卖和买不再需要同时发生，卖和买可以拆分为不同的家庭成员并行进行，分工提高了效率。在空间上，卖的时候只是获得到了货币，如果对方卖的东西不是自己想要的，交易仍然可以发生。有了通用货币，人们在买的时候可以自由选择，不再受自己所卖的东西的限制。在时间上，人们可以保留货币，留待自己需要物品的时候再购买，货币不易随时间的变化而损坏。所以，信用货币的出现，极大地提高了社会资源流转的效率。

信用货币出现之后，人们的交易生命周期发生了拆分，从交易变成了买和卖。



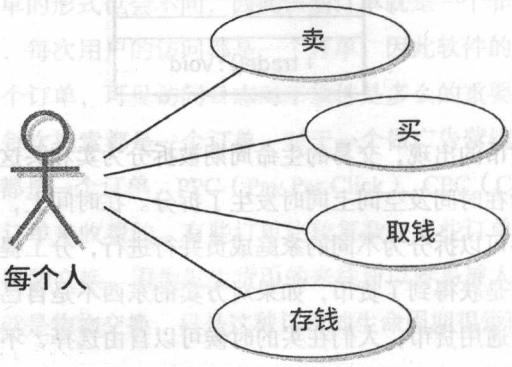
每个个体的交易生命周期，拆分成了买和卖两个生命周期。此时订单中所传递的是钱和商品，也就是所谓的“一手交钱一手交货”。生命周期仍然很短。



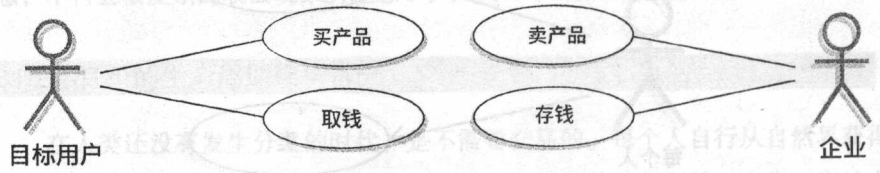
由于买和卖所用的信用货币往往是通用的，并不记名。所以为了要管理好买

和卖所需要的钱，每个个体还需要一个地方统一保存信用货币。家庭内部往往会把钱放入比较隐蔽的地方，于是保险箱就慢慢出现了，随后也慢慢出现了银行。

因此，每个个体的生命周期进一步地发生了变化，出现了存取钱的活动。



从交易主体的角度，参与交易的每个个体可以分为两类：买方和卖方。对于买方来讲，主要的行为是取钱来买卖方的产品；对于卖方来讲，主要的行为是卖自己的产品获得钱。所以就企业的核心生命周期来说，企业是通过卖自己的产品来获取自己的利益，主要以钱为代表，企业是卖方。而买方则是企业所生产产品的客户。



在现代的物流和第三方支付发展起来之后，交易中钱的传递和商品的传递又被单独拆分了出来，人们不再需要面对面进行交易。此时订单的生命周期就变得非常长了，参与方也变多了，多出了第三方支付和物流。订单生命周期就变成多方协调，所有的参与者共同推进。所以，订单实际上是各方行为的一个记录。

随着订单生命周期的架构拆分，人类的分工越来越丰富，产生了越来越多的不同企业，进一步推进了社会的繁荣。

31.3　订单支付

“一手交钱一手交货”，描述的就是通过支付一定数量的货币，来获得商品的交易过程。在货币出现后，人与人交易的主要形式是钱和商品的交换。在面对面交易时，人们基本上都是当面交货币为主。

货币的出现，导致了金融行业的出现，其中银行是代表。企业和个人的钱都会交给银行来进行管理，相当于借给银行的同时还能增值。因此，交易时往往都会和银行打交道。交易时货币的传递就变成了企业和客户的银行账号之间的传递，货币传递的路径发生了架构的拆分，因为增加了银行这个角色，所以导致了货币的数字化。

所谓支付，指的就是客户把购买商品应该付的钱付给企业。为了方便客户的支付、企业一般都有银行的服务终端，用户除了可以用现金支付以外，还可以通过信用卡，支票等方式进行支付订单的应付金额。银行的业务又有自己的生命周期和生命周期的架构拆分。

31.4　订单生命周期

订单的生命周期起始于订单的创建。订单是买卖双方达成的一致意见，卖方同意买方支付一定的成本，用来获得卖方产品的一个契约，买方指定了商品的数量、价格和运送方式等信息。

在商品交付给买方使用，并且卖家也收到买家的钱款之后，交易的双方都得到了自己想要的东西，此时订单的生命周期就结束了。或者双方都没有得到对方的东西，订单的生命周期也结束了。

第 32 章　交易系统

通过前面几章的探讨，大家已经大致搞清楚了在现实生活中，交易、用户、产品和支付之间的关系。从中也可以看到，随着人类本身对生活质量要求的提升，以及人类分工的进一步细化，人类社会的架构拆分进一步深化，不同角色的协作越来越复杂，生产力得到很大的提升。而软件的出现，进一步产生了新的社会分工，以软件来模拟人类的行为，得到了更大的生产力提升。

交易系统是企业的核心系统，如何建设好交易系统是一个很有趣的话题。通过对交易系统的分析，可以看到社会发展对软件的影响，以及软件架构是如何在软件中发挥作用的，同时软件又是如何反过来推动社会发展的。

对于交易系统，不同类型的企业各有其不同的特质。制造业类的企业以制造为主，交易往往较弱，总是依赖于渠道类的企业帮助售卖。因此渠道类的企业往往是软件虚拟化的先导，而且以交易为主项。无论是哪种企业，交易系统都是大同小异的，因为其核心生命周期都是一样的，都是服务于交易双方的。各种交易系统的区别在于企业的产品不同，售卖的流程也不一样。不同交易系统的最终目的都是为了形成一个用户访问企业产品的通道。

接下来就通过对交易系统的分析，忽略不同企业的交易系统之间行为的差异，从交易系统本身的职责拆分和架构的拆分中，展现架构是如何在软件中应用的。

32.1　企业的架构拆分

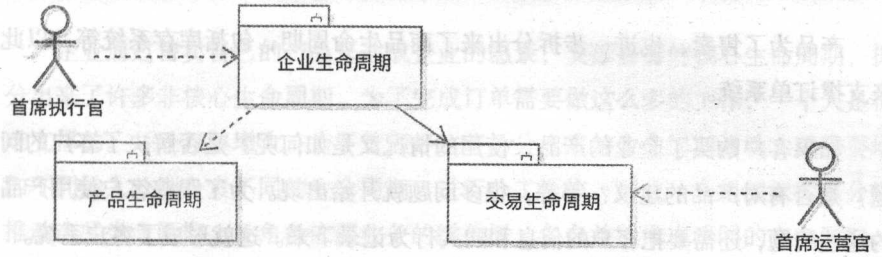
一个交易是需要两方来形成的，一是卖方，还要有一个有意愿购买的买方。要两方同时达成一致才能够形成一个交易。

对于企业来说，怎么寻找到那个对应的买方呢？这是一个很大的问题，这个问题的解决由企业本身所生产的产品来决定。企业应该生产什么样的产品呢？企业本身在社会上的分工是为了帮助人类社会解决什么问题呢？这些问题的答案决

定了这个企业的愿景，而企业的愿景决定了可以受益的人群，也决定了企业的目标客户是哪些人。企业愿景的达成依靠企业所生产产品的售卖，通过交易建立起客户独占访问产品的通道，使企业和客户都获得更大的利益，形成双赢的结果。

因此，企业的生命周期就拆分为两部分，企业的产品生产生命周期和企业的交易生命周期，其中企业的交易生命周期是核心生命周期。难道产品生产不核心吗？因为产品的生产是为交易服务的，所以很多制造型的企业往往选择代工，通道型企业的通道建立也是一样的，是为交易服务的。当然产品的生产还包括设计和建造，这里就不展开了。对比一下企业产品生命周期的架构拆分和软件生命周期的架构拆分，就可以发现两者都是类似的，因为软件也是某个企业的一种产品。

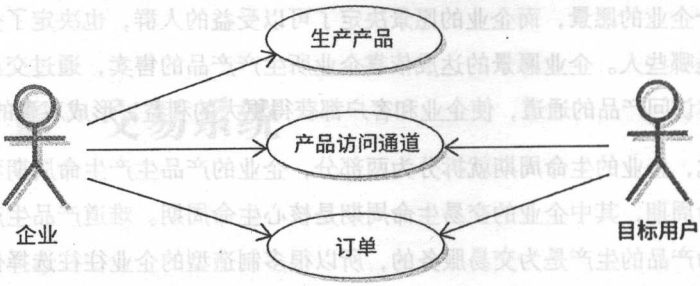
企业的产品生命周期和交易生命周期都是企业的领导人非常关心的。企业的核心是交易生命周期，这两个生命周期都会被拆分出来形成分工。由于产品生命周期往往和企业的愿景相关，因此企业的领导人会更加专注于产品。而交易生命周期则一般由企业运营的负责人来负责，为企业领导人分工。



为了和目标客户建立联系，剩下的问题就是，企业的产品通过什么样的渠道和客户建立访问通道，以便让目标客户可以访问到企业和企业的产品，产品本身又是如何展现给客户呢？企业必须把产品罗列出来，描述产品的特性，吸引用户交易。要形成订单，先要有产品。

为了记录用户的购买行为，确保用户获得产品的所有权，就需要把双方的契约记录下来。这个契约就是一个订单，有独立的生命周期。

这种关系可以用下面的图简单的表达出来。企业的核心生命周期活动包括了如下几个活动，箭头表示主体推进的方向。企业要能生产出产品，并提供产品访问通道给客户，才能够和客户形成一个最终的交易。



产品的生产不是交易系统所关心的，产品生产环节就不在此讨论。但是产品访问通道则是交易系统能够运作的前提。为了把产品展示给客户，一般就会要求产品的负责人把所有产品整理好，用来供用户查询，这就是产品系统。

一旦用户开始访问企业和企业的产品之后，企业就会开始记录用户的情况，保留用户的信息。企业的生命周期发生了进一步的拆分，形成了用户生命周期，用来跟踪用户和客户的情况。产品生命周期也因此发生了进一步的拆分，形成了营销生命周期和用户建立售前的访问通道。

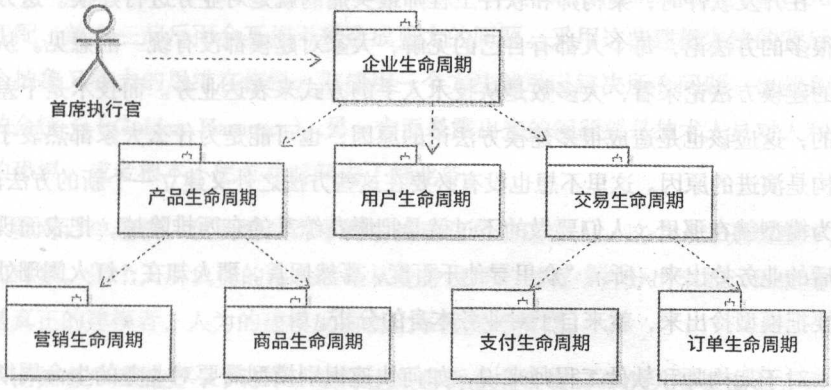
产品为了售卖，也进一步拆分出来了商品生命周期，包括库存系统等，以此来支撑订单系统。

如果客户购买了企业的产品，使用的情况又是如何呢？是否解决了客户的问题？是否有对产品的建议？等等，很多问题就开始出现。为了跟踪客户使用产品的这些问题，还需要把客户的信息和购买行为记录下来。这就形成了客户系统。

有了用户的购买行为和联系方式，客户的通道就建立了。利用这个通道就可以在适当的时机推荐合适的产品，或者在客户再次购买的时候，快速的帮助客户选择到更合适的产品，等等。这就是营销推荐系统。

客户和企业之间的金钱来往，也需要独立的管理，也就是支付体系。渠道型的企业往往要有供应商的支撑，因此和供应商也有金钱的来往。所以会有应收应付的问题，往往需要清算（Clearance）的支持。企业本身的钱，也会统一交给银行来管理，因此企业也会和银行打交道，会有清算系统或者会计系统等来支撑。

所以一个企业的架构拆分如下图所示，其中交易生命周期和订单生命周期是核心生命周期，而产品、用户等生命周期都可以不用自己亲自来做，但订单是非要自己来做不可的。比如很多互联网企业可以采用第三方用户认证，产品可以外包，营销可以找第三方公司等都是这个原因。认识清楚了企业的核心生命周期，就可以在建立企业的过程中有的放矢，把资源用在刀刃上。



企业通过售卖自己的产品来达成企业的愿景，支撑自身的核心生命周期，拆分出来了许多非核心生命周期。为了完成订单需要做这么多的工作，一个人是很难支撑的。一旦订单增大，为了管理好这些拆分出来的生命周期活动，就需要增加不同的人分别负责不同的生命周期，进行分工合作，形成一个组织架构来分别推动这些生命周期。各角色依照业务的拆分树，各自并行推进不同的生命周期，有人负责整个公司的运营，有人负责营销，有人负责产品，有人负责交易，有人负责支付和结算等，这些人把各自生命周期的推进结果汇合到树的根节点，共同为企业生命周期服务。由于生命周期的拆分是树状的，因此组织架构也同样是树状的架构。

拆分之后，企业领导人的工作从原来的什么都干，到逐渐把企业的运营工作下放到了组织架构树下的不同角色身上，主要时间花在思考企业的愿景，企业在人类社会中的定位上，为企业的战略服务。

有了这些分工的支撑，企业的生命周期就得以顺畅运转。随着营销的覆盖面扩大，越来越多的大众开始了解企业的产品；由于产品本身的吸引力，越来越多的人购买产品，成为企业的客户，交易量越来越大；随着各生命周期的壮大，交

易生命周期得以迅速流转；交易量增大，企业也开始变得越来越大。随着每个角色所负责的生命周期本身的活动增多，慢慢也会形成新的瓶颈，越来越多的架构拆分也开始发生，整个企业的组织架构树也逐渐长得更加高大。

32.2 软件系统的建模

在开发软件时，架构师和软件工程师最头痛的就是对业务进行建模。这方面有很多的方法论，每个人都有自己的见解，大家对建模都没有统一的意见。从已有的建模方法论来看，大多数是从技术人手的方式来表达业务。而技术是千差万别的，这应该也是造成很多建模方法论的原因，也可能是为什么大家都热衷于说架构是演进的原因。这里不想也没有必要在这些方法之外又建立一个新的方法论，因为模型就在那里。人们要做的不过就是把眼花缭乱的东西排除掉，把表面现象背后的业务拎出来。所谓“众里寻他千百度，蓦然回首，那人却在，灯火阑珊处。”而要把模型拎出来，就来自于对业务本身的分析。

对于架构师和软件工程师来说，如何快速识别模型需要对业务的生命周期进行深入了解，要对业务的特性有足够的认识和体会。这就是为什么前面用了很大的篇幅，对业务的生命周期的拆分和业务的历史演变等做深入的分析，确保我们理解了业务的背景和痛点，也就是业务的个性。而业务背后是人的行为及行为背后人们的利益诉求，这一点是绝对不能够被忽略的。架构拆分历史的背后是参与人的利益受到了限制，所以需要拆分来打开瓶颈。

对业务系统的建模来源于业务本身生命周期的拆分，即组织架构的拆分。没有分工的支持，业务本身是不可能被拆分出来的。我们已经知道，软件是对人的模拟，而建模实际上就是把人虚拟化，把各业务系统负责人的工作用模型表达出来。模型中的每一个对象，代表的就是现实生活中一个个活生生的人。这个对象的方法，代表的就是这个人所负责的业务生命周期的行为。

比如我们所说的订单，订单本身是一个完整的生命周期。其生命周期是不会自己推进的，在企业内部会有负责订单的人来专门推动订单生命周期。订单其实是负责订单的人推动订单生命周期过程中每个订单活动的数据记录，是每个活动的状态、结果的数据积累。

大部分做技术的人往往忽略了人，而只看见了订单这个数据，拿数据来建模。有些人看见了数据的变化，把它认识成为一个状态机，用状态机来建模；有些人看到了这些状态都是操作的结果，对操作来进行建模等。还有很多其他的建模方式。这些建模方式都有其自己特定的业务领域，但是这些特定的业务领域和订单的业务领域并没有关系，也无法表达订单背后的业务，或者只能描述某方面的特质。这就是所谓的“阻抗不匹配”（Impedance dismatch），也就是说两个东西完全不匹配，放在一起反而会互相干扰造成更大的问题。采用这些建模方法的背后是一个抽象又省力的思维在作怪，期望用一个方法就可以解决所有问题，也就是所谓的金锤子（Golden Hammer）。另一方面暴露出来的问题就是技术人员对人和业务的恐惧，或者根本不愿意花时间去了解业务。

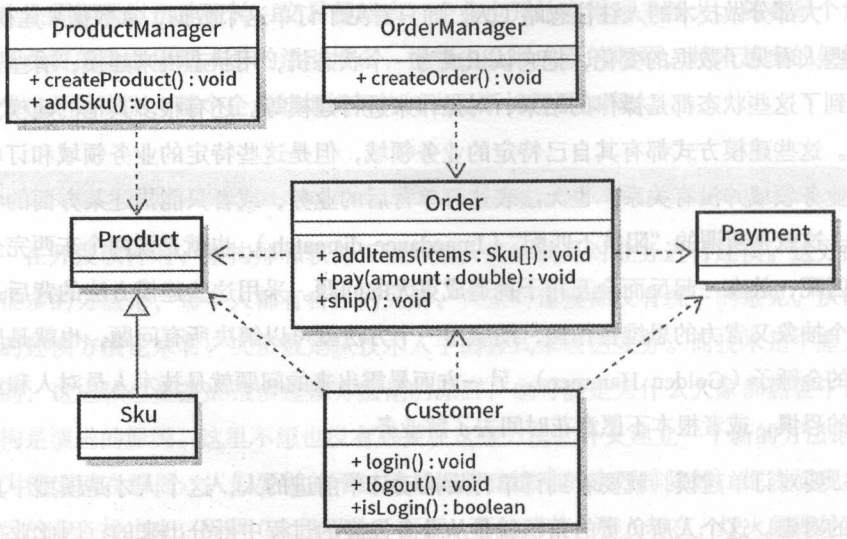
要对订单建模，就要看到订单背后推动订单前进的人，这个人才是模型中真正的对象。这个人所负责的角色就是从业务发展的过程中拆分出来的，因此业务才是真正的建模者，人为的建模反而会让系统变得糟糕。

比如客户要提交一个订单，实际上是告诉企业负责订单的人要买商品，由负责订单的人来创建一个订单把客户的信息记录下来，然后接受客户的付款，再把所购买的商品传递给客户。这是一个典型的订单生命周期。

订单负责人是怎么出来的呢？是在企业的组织架构分拆过程中出来的，组织架构拆分则是为了对业务进行分工，利用不同角色并行工作共同推动业务，来提高企业生产力的。订单实际上是企业订单负责人推进订单过程的一个记录，订单的操作是订单负责人处理订单的行为，行为产生了数据并记忆在脑中，是订单负责人的记忆。人的脑容量有限，记不住那么多订单，所以就用文件的方式把订单记录到了文件中，于是订单的模型就出来了。

所以面向对象是一个非常好的表达方式，把行为和记忆结合了起来，完整模拟了一个现实生活中的人。大部分架构师和软件工程师把面向对象看成是一种技术，而不是对现实生活的模拟和表达，这就太遗憾、太可惜了。

所以根据以上的分析，可以得出如下的订单业务模型：



Order 代表的就是处理订单的负责人，需要根据用户的要求做相应的动作，比如往订单中加入商品 SKU（Stock Keeping Unit）、付款等。而订单是无法自己给自己创建的，因此订单的负责人还需要一个角色来负责创建订单，也就是 OrderManager，用来管理订单的生命周期。OrderManager 和 Order 之间的关系就是：Order 的意义就好比是一个真正执行订单生命周期的人，是干活的。而这帮干活的需要有一个人来协调管理，统一对外服务，也就是 OrderManager。这就是现实生活中组织架构在软件建模中的映射，对于 Product、ProductManager 也是同样的道理。

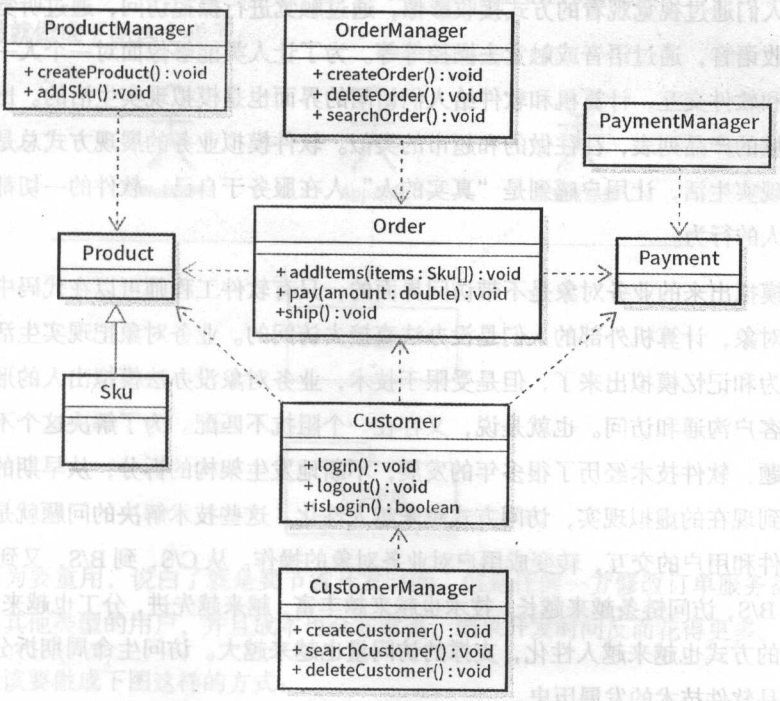
业务模型完全就是现实社会中已存在的分工，建模的人只需要识别出来就可以，不需要再绞尽脑汁用不同技术抽象出来另外建一个模型来表达业务。

对于软件系统来说，由于软件没有意志，软件自己是不会主动发起行为的，发起行为的是坐在计算机前面的人类。当然，在触觉等感觉也都被虚拟化了之后，人或许就可以生活在计算机里了。所以每个角色都在软件外面的现实世界中访问软件里面虚拟化自己的那个对象，操控他们来完成自己的工作。对于企业的内部人员，在没有软件之前是要亲自执行订单中的所有工作的，用户通过发起指令要求企业内部负责订单的角色做相应的订单操作。有了软件之后，负责订单的角色就把订单的操作交给软件中的对象来代替自己完成这些工作。对于客户而言，原

来是直接面对企业的内部人员下订单的，现在就直接面对软件中的对象来提起自己的请求，不再经过企业内部人员的转达，彼此的沟通变得更加自然，企业内部人员的工作负担也得到了减轻。

对于软件来说，谁都可以访问这些对象。那么这些内部对象如何识别出是哪个客户的操作，对应的是哪个订单呢？这就地求每个操作都需要带上客户的标识，表明订单的归属方。而要带上这些标识，就必须要求客户在操作之前先登录，而要先登录，就必须要先创建这个用户。为了管理好用户，企业还需要提供一个客户经理。至此，用户和企业的相关角色都模拟出来了，双方共同推动产品、订单的生命周期向前推进。

这些对象就好比是一个一个的机器人，在企业方操作软件的人是控制机器的，在操控不同的机器人来帮自己完成工作。而客户则变成面对机器人提出要求，完成对企业服务的访问。这就是面向对象的实质，也是面向对象的精髓。理解了这一点，面向对象就会发挥出巨大的能量，把现实世界忠实地模拟出来。如下图所示，虚线箭头代表访问。



看到这个图，很多人会有疑问，为什么不是树状关系呢？因为图中表达的是不同的角色进行合作的互相依赖关系，会包含有多条访问路径。所有的访问路径合并起来就形成了一个有向图。但是针对单条访问路径仍然是一个树状的架构。如果对象之间的依赖图形成了一个无向图，就可以知道生命周期拆分发生了问题，对象的拆分发生了问题。

32.3 访问业务模型

在业务模型建好之后，现实生活中的各个角色，如何访问业务对象呢？解决这个问题就靠软件访问生命周期来实现。在前文中已经探讨过软件访问生命周期的演变历史，这里就不再重复。

为了让各角色能够访问到这些业务对象，即业务机器人，就必须提供给人们可以访问的方式。人类能够访问的方式无非就是这几种：视觉、听觉、嗅觉、味觉和触觉等。目前能够虚拟出的也就是视觉和听觉的对象，但未来可能会有嗅觉、味觉和触觉的感知对象被模拟出来。

人们通过视觉观看的方式接收影像，通过触觉进行操控访问，通过听觉的方式接收语音，通过语音或触觉去操控等等。为了让人类能够像面对一个人一样自然地和软件交互，计算机和软件给人们访问的界面也是模拟现实生活的。比如网上商城的产品列表，往往做的和超市的类似。软件模拟业务的展现方式总是尽量模拟现实生活，让用户感到是“真实的人”人在服务于自己。软件的一切都是在模拟人的行为。

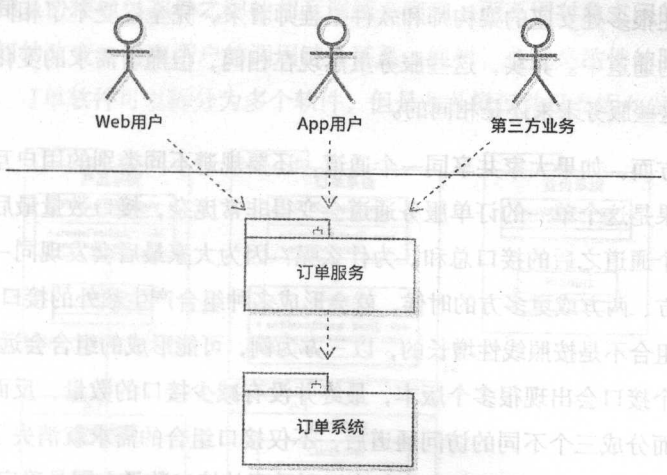
模拟出来的业务对象是不带访问界面的，只有软件工程师可以在代码中调用这些对象，计算机外部的人们是没办法直接去访问的。业务对象把现实生活中人的行为和记忆模拟出来了，但是受限于技术，业务对象没办法模拟出人的形象直接让客户沟通和访问。也就是说，又存在一个阻抗不匹配。为了解决这个不匹配的问题，软件技术经历了很多年的发展，不断地发生架构的拆分，从早期的字符界面到现在的虚拟现实，访问方式越来越人性化。这些技术解决的问题就是怎么把软件和用户的交互，转变成用户对业务对象的操作。从 C/S，到 B/S，又到 C/S，又到 B/S，访问链条越来越长，技术也越来越丰富，越来越先进，分工也越来越多，访问的方式也越来越人性化，支撑的访问量也越来越大。访问生命周期拆分的历史就是软件技术的发展历史。

软件业务和软件所服务的业务是两种不同的业务，因此就需要分别使用不同的业务模型，不能够混杂在一起。这就是为什么访问代码里面不能混入业务模型代码的原因。访问的阻抗匹配问题的解决，形成了很多的设计模式，设计模式就是技术。

访问通道的另一个非常重要的要求就是，不同类型的用户访问要提供单独的访问通道。

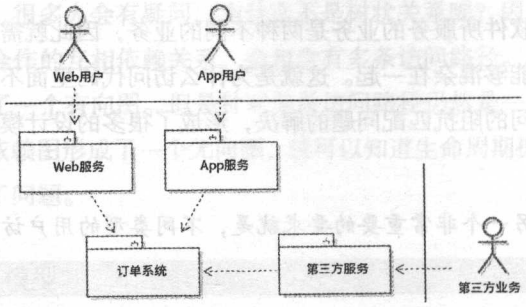
比如人们登机，虽然都是同一趟飞机，但是有 VIP 通道、普通旅客通道、贵宾候机室和普通候机室，目的是避免不同的角色相互影响对方的访问。很多架构师喜欢做访问通道的重用，这是一个很严重的错误，并且会导致非常大的问题。因为一类用户访问的变更，往往会导致不相干的另一类用户访问受影响。受影响的另一类用户既委屈又无奈。当多种用户共享的时候，影响面就更大了，通道修改起来也就越发困难。

受软件重用的毒害，架构师和软件工程师把重用当作软件代码质量评判的至高无上的标准。任何事情如果极端化，都会产生反面的作用。比如一个订单服务，很容易就做成下图这个样子：



因为要重用，说白了就是要节省开发时间。但是任何一方修改订单服务都会影响到其他类型的用户，并且成本也会非常高，结果开发时间反而花得更多。

应该要做成下图这样的方式：



要让各方都有自己的访问通道，保证访问通道不能够重用。

因为通道的作用是做访问通道的阻抗匹配的，便于访问到业务模型。每种类型用户的访问方式都有不一样的特点，无法用一个通道来覆盖所有用户类型的特征。

不重用访问通道的目的是为了重用业务对象。一旦重用访问通道，业务对象中的业务逻辑一定会分散到访问通道中，业务逻辑反而得不到重用了。

重用访问通道还是重用业务逻辑，二者只能选一个，无法兼得，毫无疑问要选重用业务逻辑。

有人会有疑问，这样不是服务就变得很多了。确实是的，甚至会有很多相同的接口。在很多处女座的架构师和软件工程师看来，完全忍受不了相同的服务出现在不同的通道中。其实，这些服务虽然现在相同，但随着需求的变化和修改，并不代表这些服务未来还是相同的。

另一方面，如果大家共享同一个通道，还要规避不同类别的用户互相影响，最后的结果是这个单一的订单服务通道会变得非常庞杂，接口数量最后会远远超出分成三个通道之后的接口总和。为什么呢？因为大家最后会发现同一个接口在服务于一方、两方或更多方的时候，就会形成多种组合产生额外的接口组合需求，并且这些组合不是按照线性增长的，以三方为例，可能形成的组合会远远超三个。导致同一个接口会出现很多个版本，最终并没有减少接口的数量，反而增加接口的数量。而分成三个不同的访问通道后，不仅接口组合的需求就消失了，而接口数量也不会疯狂上涨，因为每个角色访问所需要的接口数量上限是确定的。

对于内部访问通道的处理也是同样的规律。比如一个部门要服务多个其他部门，那就要给不同部门以不同的服务通道，避免不同部门的访问通道互相干扰。究竟应该分为几个通道比较合适，还有一个判断技巧，就是看有几个不同的产品

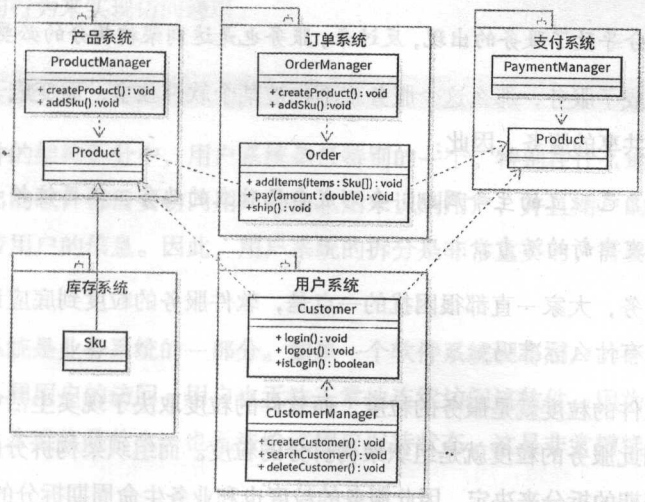
经理。针对不同的产品经理提出的需求，要分别给不同的服务通道。因为需求都是从产品经理发出来的，某个需求修改的影响在这个产品经理范围内是可以得到预估和协调的。

32.4　交易软件系统的架构拆分

当企业组织架构拆分之后，每个部门往往就会独立的提出自己的需求，针对自己所负责的业务生命周期进行修改。如果把交易系统做成一个软件，就容易造成不同部门争抢软件开发资源。同时，软件也容易变得非常庞杂，运行的效率也会下降。因此，软件本身也会逐渐地发生架构拆分。

软件本身的拆分，是由企业的组织架构拆分导致的，因此软件本身也会按照组织架构拆分的方式，把不同的业务对象拆分到不同的软件中。拆分的原则也是和组织架构的拆分保持一致的。

拆分的结果，产品系统、订单系统、支付系统、库存系统和用户系统都分别独立出来，形成了新的软件。原来在同一个软件里的时候，不同对象之间的互相协作是在软件内部通过互相的调用来实现的。当拆分为不同的软件之后，不同对象之间的相互协作通过软件之间的相互调用来实现，而不同对象之间的模型并没有发生任何的改变，每次用户的调用链条还是一棵树，这就是软件的服务化。如下图所示，订单软件可以拆分为多个软件，但是业务模型并没有因此发生改变。



32.5 服务的产生和粒度

什么是服务呢？服务是对英文 Service 的翻译，指的是一方帮助另一方解决某种问题。比如在现实生活中访问银行的营业厅，营业厅里的柜台窗口就是一个服务，这个服务就是访问银行的通道。在计算机的 C/S 架构中，服务一般特指服务端所提供给客户端调用的功能集合，用来表示与本地调用的区别，虽然本地调用也可以看成是一方对另一方的服务。

在计算机软件中，服务的本身是客户端访问服务端的访问通道，是不包含所服务的业务的。比如客户要在银行柜台存钱，这个钱不会被柜台的服务人员带回家，柜台服务人员也不负责给客户利息。柜台的服务人员本身并不能提供银行的服务，而只是作为用户访问银行核心服务的通道。

银行为什么需要服务人员来提供服务呢？因为银行的业务不仅有自己的特质，还有自己的专业语言，这些特质和语言并不为普通的用户所理解。服务人员通过把用户的请求，转换成对银行业务的语言访问银行业务来完成用户所要求业务的执行。也就是说，服务人员起了一个阻抗匹配的作用，提供了一个更好的用户访问界面。

一旦软件发生了架构拆分，原来的本地调用就变成了服务调用。

不同的服务形成了不同的软件，不同的软件可以并行的进行管理。但是运行时，通过服务调用形成的执行还仍然是串行的。

架构拆分导致了服务的出现，反过来，服务也是达到架构拆分的必要技术手段。

一旦形成了服务，那么这个服务就不再被某个软件独享，而变成了一个其他软件都可以共享的服务。因此：

服务有自己独立的生命周期，会形成自己独立的业务，不再依赖于原核心生命周期，焕发出新的活力。

对于服务，大家一直都很困扰的一点是，软件服务的粒度到底应该是由什么决定的呢？有什么标准吗？

每个软件的粒度就是服务的粒度，而软件的粒度取决于现实生活中组织架构的拆分，因此服务的粒度就是组织架构拆分的粒度。而组织架构拆分的粒度又由业务生命周期的拆分来决定，因此服务的粒度也和业务生命周期拆分的粒度有关。

一个服务软件包含多少个业务生命周期，取决于该软件所服务的业务组织负责管理多少个业务生命周期。并且这些业务生命周期往往都有相关性，是树的一部分。

一个服务软件最少要包含一个业务生命周期，这是最小的服务粒度。在现实生活中，有些生命周期是必须连续发生，不可拆分的，这是最小粒度。比如孕妇怀孕生子，十月怀胎是不可分割的，不可以分成十个人每人一月来完成。架构拆分最终都会形成最小粒度的生命周期，粒度有多小取决于业务本身的规律，也取决于当前技术的发展水平。如果业务本身生命周期的进程就不可拆分，这就是最小粒度。如果业务本身的生命周期可以拆分，但是受限于技术无法拆分，这也是最小粒度。比如在第三方支付保障（Escrow）技术出现前，网上购物就是不可能实现的，订单的生命周期就无法突破“一手交钱一手交货”的模式，订单生命周期的最小粒度就是“一手交钱一手交货”。第三方支付保障技术出现后，“一手交钱一手交货”的模式就被打破了，不再是最小粒度，形成了新的拆分，产生了更细粒度的业务生命周期。再比如未来试管婴儿技术的发展，可能会突破十月怀胎的限制，那时十月怀胎就不一定是拆分的最小粒度了。

服务的目的，就是为所服务的用户提供服务所包含业务生命周期的访问。服务要把背后业务生命周期推进的行为暴露出来，按用户能理解的方式进行组合，供用户调用访问，以便操作和控制“业务机器人”，获得模拟出来的业务人员的服务。同时还要提供访问方式，使得用户可以获取业务生命周期本身的内部状态，也就是“业务机器人”的记忆。因此服务本身是没有业务逻辑的，只是组合业务生命周期的行为来实现访问通道。

32.6　用户系统的拆分

在软件的架构拆分中，用户系统是最特别的一个。特别在什么地方呢？任何一个拆分出的软件都需要访问用户的信息用来识别用户，并且用户的每一次访问都需要附带用户的信息。因此，用户系统的拆分是非常重要的，需要拿出来单独讨论。

用户系统是业务系统的一部分。如果一个软件系统没有用户的标识，那么就无法区分不同用户的访问。用户也无法正常地连续访问该软件，因为软件无法知道用户的上个操作是什么，也无法给该用户保持状态。这是非常糟糕的。在现实

世界就不存在这个问题，因为现实世界人的访问要动用自己的身体，这个身体本身就是一个标识，另外一个人可以通过视觉、听觉等感觉来对人做识别。软件系统只能通过生成一个标识来模拟一个人，并把这个标识放在用户的访问信息中让软件来验证，这就变得和现实世界一样了，完成了对现实世界的模拟。

在用户系统作为单独的软件拆分出来之后，其他的软件如订单系统、产品系统没有了用户系统的支撑，发现既无法识别用户也无法区分用户的访问了，这时应该怎么办呢？有必要分析一下。

在用户系统中，用户是现实生活中活生生的人在软件系统中的一个映射。从用户的定义来看，用户是使用软件的人。由于软件是模拟业务的，所以用户同时也是使用软件所模拟的业务的人，具有双重的含义。因此，用户的生命周期开始于用户第一次对软件的访问，在用户不再使用该软件或该业务时为结束。

当用户第一次访问软件时，用户就应该被生成，用来跟踪用户对软件和业务的访问。生成的过程可以是显式的，此时需要用户填写个人资料、创建个人密码等。如果不强制用户注册，那么生产的过程就是隐式的，对用户透明，避免打扰用户的访问过程。比如生成一个唯一标识，和用户的设备绑定等。采用隐式标识的用户不需要登录，因此大多数时候只能够用于浏览。

在用户显式注册时，用户要提供自己的个人资料，并生成用户名和密码来保护自己的身份。此类用户在使用软件时，需要显式的登录来证明自己是声称的那个用户，这就是登录。用户为了防止自己的身份被冒用，使用完软件之后还需要登出。登录和登出的两个动作，表示用户一次登录的生命周期，在登录和登出之间，用户的行为都会被记录在该用户的名下。在一个用户的生命周期中，用户的登录生命周期可以发生多次。

在现实生活中，用户的行为是非常难记录的。比如在一次足球比赛时，每个球员的触球、抢断、奔跑、射门等动作是做球员分析的好数据。这些数据每天、每时、每刻都在发生，由于大自然的特性，不会自动保留这些信息。为了要记录这些数据，人们不得不盯着画面，对每个队员，每个动作做人工的记录，成本非常得高。也许当视觉模拟的技术足够成熟后，我们可以把人们的每一个动作都搜集下来，未来的监控会越来越发达。

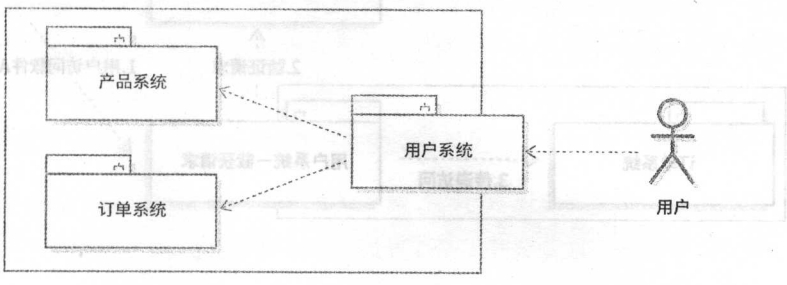
但是在虚拟世界里，用户的每一次访问都可以用软件记录下来，即用户的访

问日志。用户的访问日志里面包含了用户对软件的所有访问信息，对业务的所有访问信息。通过对这些信息的分析，不仅可以帮助软件提升，还可以帮助业务提升。因此，在访问中标识用户是非常重要的。

从用户的生命周期分析中可以看到，用户系统的核心是为了能够标识用户，让用户能够以独立的身份访问系统，从而软件可以识别并跟踪用户。而用户的生成和登录，则是达到这一目标的必要路径。访问软件则又是用户访问软件生命周期的核心。可见用户的生命周期管理是服务于软件的访问生命周期的。我们所说的软件访问生命周期指的就是用户访问软件的生命周期，因为软件是为人类服务的，访问的主体是用户。由此可见，用户的生命周期其实就是软件的访问生命周期中的一部分。因为发生了架构拆分，用户软件被拆分出来，导致用户的生命周期从软件访问生命周期中拆分了出来，使得其他软件无法识别用户了。

如果不拆分的话，每个软件都需要把用户的生命周期纳入到各自的软件开发中，会产生很多问题，如用户的同步、开发的浪费等。把用户生命周期从其他软件的访问生命周期中拆分出来是必须的，但是可以不把用户从访问生命周期中拆分出来，软件访问时只需要带一个标识即可代表用户。这样用户生命周期的管理就可以独立于其他软件访问生命周期的管理，达到空间和时间上的并行，只需要保证其他软件的访问生命周期时间上的执行顺序不变即可。并且其他软件不用再管理用户的生命周期，因此内容就得到了精简，变得更加专注。

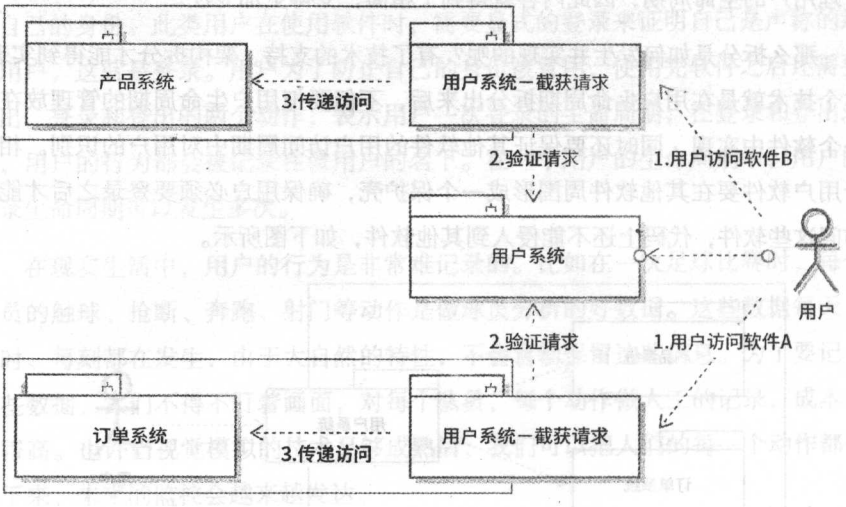
那么拆分是如何发生并实现的呢？有了技术的支持，架构拆分才能得到实现。这个技术就是在用户生命周期拆分出来后，不但要把用户生命周期的管理放在另一个软件中实现，同时还要保证其他软件的用户访问周期中对用户的识别。相当于用户软件要在其他软件周围形成一个保护壳，确保用户必须要登录之后才能够访问这些软件，代码上还不能侵入到其他软件，如下图所示。



更进一步，如果能够做到在某个软件中登录之后，虽然软件已经发生了拆分，但在其他软件中仍然不需要再登录，通过所谓的 Session 共享，给用户造成使用同一个软件的感觉就成功了。用户系统软件就相当于是一个黏合剂，把拆分的软件重新黏合起来，给用户的感觉好像是在访问同一个软件。

由此可见，内部架构的拆分并没有造成业务的分裂，也没有对软件访问生命周期造成影响，软件访问生命周期中活动的先后顺序和拆分之前是一样的，只是部署不一样。也就是说管理是并行的，但是运行时的访问顺序并没有变。这同时也证明了，做架构的拆分不会影响外部。而影响外部访问行为的拆分，都是对业务的破坏，不应该叫作架构拆分，而应该叫作结构变化或业务变化。

用户软件是如何做到保护壳的作用呢？要达到这个作用，在用户的请求到达目标软件的功能之前，必须先要截获用户的请求，并把请求交给用户软件做用户身份的检查。要实现这一点，用户软件就必须要提供某种形式的服务，供所有的其他软件来调用，同时还要提供一个客户端帮助其他软件截获用户请求，并调用用户服务来验证。所形成的系统如下图所示，用户系统被分为两部分：一部分提供用户本身的服务，如注册、登录等，另一部分部署在其他软件的前端，负责在用户的请求到达其他软件之前，截获用户的请求，验证用户的身份。这部分往往被称作统一用户登录系统。



为了防范恶意的用户注册或登录，还需要做相应的安全措施，这里就不一一展开。

对用户的生命周期的管理及用户的活动分析管理，也都属于用户系统的范围。业务部门往往会有分工，根据用户生命周期分工的不同，会形成客户关系管理部门、用户注册登录部门等。会导致用户软件被进一步的拆分为用户的注册管理软件、用户的登录管理软件和用户个人中心等。

第33章 事务

一旦软件被拆分，或者服务被拆分，或者数据库被拆分，很多架构师和软件工程师的第一反应就是事务（Transaction）怎么保证？一旦说事务，基本上大家马上条件反射地认为是指数据库事务，背后的含义也就是指数据库的 ACID 特性：原子性（Automicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

这是当前给架构师和软件工程师带来大量困难的习惯，并且非常的普遍，思维基本上被关系型数据库绑架了。背后反映的是大家平时对数据库事务的依赖，以致于把数据库的特性本身就当作了业务的一部分，结果所有的业务思考全部都要放在数据库的前提下。问题的实质其实就是软件如果不拆分，两个操作会在同一个软件下，两个操作执行的一致性就可以用数据库的事务来保证。而一旦两个操作被拆分在两个不同的软件下，数据库事务就用不上了，拆分就无法落地实现。除非采用分布式事务，而分布式事务又是一个大坑，会极大地降低性能。因此在这个前提下，架构师和软件工程师往往反对软件的拆分，因为没有了数据库事务的支撑就实现不了架构拆分，更别说数据库的拆分了。

在很多人的脑海里，软件就是：数据+对数据的操作，很多应用更干脆直接把代码写在数据库中。结果应用开发是简单了，但是一旦访问量增长，就会发现瓶颈就在数据库上，于是数据库爆了。最终只好换更好更强大的计算机，只能够纵向扩展。数据库横向扩展可不可以呢？对不起，由于应用模型一开始就是依赖于数据的，因此数据访问是怎么方便怎么来。而数据库采用行列的数据组织方式，同一个表中会包含所有用户的信息，因此单表的数据是很难拆分的。数据库不同的表之间也会有大量的 Join，因此不同的表也要在一个数据库软件才能跑起来。

很多人说，这些我改还不行吗？确实还真不行，因为问题的背后就是数据库事务的好处带来的，这是软件工程师等技术人员的利益所在：不管多少个业务操

作，只要一个事务提交就完成了。数据库拆分之后，事务如何保障？由此可见，要打破大家对事务的的理解才是根本。否则不管学多先进的编程语言和多先进的软件开发的理念，碰到数据库事务全都会现出原形，最后写出来的应用还是：数据+对数据的操作，还是需要靠数据库的事务保障。

很多公司在招人面试时，数据库事务基本是必考项目。数据库思维对整个行业影响要比我们想象中大。

既然事务这么重要，我们先来看看什么是事务。

33.1　什么是事务

事务是英文 Transaction 的翻译，并非中文事务的本意。在中文中，事务是指某个具体的事情或者杂务。计算机软件行业中，很多英文词汇的翻译和中文的原意差距太大，导致的后果就是软件从业者所用的词汇无法被大众理解，形成了巨大的沟通障碍，可惜可叹。

在英文中，Transaction 的含义其实是交易的意思，指的是物物交换。在信用货币时代，指的是“一手交钱一手交货”，交钱后拿到货，钱货的交换要都达成，使得交易的两方都能够的到好处，也就是双赢。

因此可以看到，事务并非数据库发明的，也并不只存在于数据库中，也不是数据库专用的词汇。分工发生后，交易就普遍存在于人类的现实生活中，因此最早出现 Transaction 的行业就是商业领域。比如金融事务（Financial Transaction），有物品的交换，有货币的交换等。

做交易时，买方和卖方相互要进行交换，买方和卖方都要得到对方做交换的物品。比如买方花 100 元购买卖方的一本书，交易要么成功，也就是买方得到卖方的书，卖方得到买方的 100 元钱；要么失败，也就是买卖双方没有发生交易，各自仍然拥有各自的商品。如果买方拿了书而没付钱，或者卖方拿了钱而没给书，这就不是交易了，这是欺骗或者抢劫。对于这类意外的处理，受损失的一方会主张权力，追溯自己的财产。如果无法自己行驶权力，往往会依靠第三方来协助，比如执法机构、法院等。

新技术，也就是第三方支付担保的出现，把交易的生命周期拉长了，使得买

卖双方不需要面对面就可以进行交易。通过物流传输物品，通过第三方支付担保传输货币，达到物物交换，即双赢的效果。仍然是“一手交钱一手交货”的效果，只不过交换的过程变的非常长，交易的环节也就是交易的生命周期活动增多了而已。如果买方没在规定的时间内收到卖方发出的货物，第三方支付担保会把钱退还给买方。

33.2 软件中的事务

在现实生活中，人们做交易时，会在脑海里记住交易的历史：自己给出了什么，从对方得到了什么，以及当前的状态是什么。每个人会根据当前的状态自行采取相应的行动。

而在软件中实现交易的时候，一般是通过订单来记录各方的当前状态。交易中过往行为的记录，则是通过操作日志来实现的。操作日志和订单记录结合起来模拟人类大脑的记忆。万一丢失当前的状态，还可以通过操作日志来恢复。所谓的回滚，也是通过已执行操作的日志记录，按发生顺序反向执行恢复反向操作来进行的。因此交易日志在软件中是至关重要的。

前文已经提到，数据库在软件中扮演的作用，相当于是实现大脑的记忆存储。假设计算机本身的内存容量无限，即使停电内容也不消失，那么软件在运行时不需要存储的介入，自己就可以玩转了。但是目前计算机还达不到这个技术，在计算机中，内存易损失，并且容量有限。所以要把软件的状态存储从软件的运行生命周期中拆分出来，形成了软件的存储外置的现状。为了能够把软件的状态持久化，要在用户的访问结束后，把每一个对象的生命周期状态的改变及时地持久化保存在计算机之外的存储中。存储有很多种，数据库是其中最典型也是最流行的可靠存储。人类在记不住事情或者要精确记录事情的时候，一样会采用外部存储，这并没有什么区别。

数据库引入软件中后，可以认为软件和数据库之间发生了一个交易：软件把内部运行中的生命周期状态交换给数据库来保存，而数据库本身获得了生存，有了自己的生命周期，形成了新的存储技术。而为了确保把软件的状态可靠的存储、可靠的获取，就需要一定的机制来保障这个交易，这就是数据库的 Transaction 技术，也就是所谓的事务。为什么数据库把这个技术叫做 Transaction，应该就是要

达到可靠数据交换的缘故。数据库为了达到数据交换的可靠性，自然也要靠事务日志来实现，也就是所谓的 Transaction Log、Transaction Journal 或 Binary Log。明白了交易的机理和生命周期的拆分，数据库事务的实现也就不难理解，为什么数据库对数据存储的保障也叫事务也就可以理解了。

33.3　数据库事务的滥用

数据库提供了这个能力之后，对于开发人员来说是一个非常大的好处。但是随之出现了数据库事务的滥用。我们以两个用户之间的转账业务操作来举例，举这个例子的原因是因为几乎所有人都用这个例子来解释数据库事务的概念。

什么是转账呢？用户 A 转账 100 元给用户 B，用户 A 的账户减少 100 元，用户 B 的账户增加 100 元。这两个动作要么都发生，要么都不发生。而只有其中一个发生，就是非法的。这就是转账的要求。从转帐的定义来看，和交易中双方发生的交换，要么都发生，要么都不发生是一样的。这样的现象都可以被称为 Transaction，也就是事务。

从账户 A 到账户 B 的转账，要先减掉账户 A，才能给账户 B 增加。一般采用复式记账的方法，把各账户资金变化的状况记录下来，类似于交易日志。

可以看到，转账是有自己的业务生命周期的，从账户 A 转移到账户 B 有自己的一套业务方式，有着自己的生命周期切分。但是很多人不去了解复式记账，把转账的技术实现完全交给了数据库的事务。也就是说数据库的事务机制承担了两个机制，一方面要保证把软件的状态也就是记忆持久化；另一方面还要保证业务的事务得以实现。后一个机制超出了数据库本身的能力，数据库事务被滥用了。当滥用变成了通识，软件行业的从业人员就很痛苦，整天挣扎在数据库的限制中。

反过来思考，当用户 A 的账户在银行甲，用户 B 的账户在银行乙时，因为银行不同数据库也不同，此时数据库事务就不能采用了，那么转账怎么实现呢？转账还是要靠复式记账本身的机制来实现，有金融行业自己的领域模型，和数据库并没有关系。

33.4　数据库的正确使用方式

数据库应该完全和软件所模拟的业务脱离关系。数据库应该恢复到其本身的

作用上来，即持久化软件的状态，并且仅仅持久化软件的状态。软件所模拟业务的自身事务，应该自行实现，不能依赖于数据库的事务。想通过数据库的事务来实现业务的事务，结果往往会适得其反，最终会影响到软件和服务的架构拆分。软件的架构拆分出了问题，就会导致软件无法随着业务的长大而长大，反而变成了业务的瓶颈。

回到转账这个例子，如果账户 A 和账户 B 的转账业务有自己的事务模型，不再依赖于数据库，账户 A 减 100 元的操作和账户 B 增加 100 元的操作就不用在同一个数据库事务里面，软件对数据库的操作就非常简单了，单表访问即可。数据库的事务也变得简单了，只需要保证单表的事务，数据库的横向扩展也变得非常的容易。如果只需要单表事务，那么就不仅仅是关系型数据库可以选择了，还有更多更好的技术可以帮助软件实现业务生命周期状态的可靠持久化。

33.5 服务调用

回到最初的问题，用户要调用的两个目标服务被拆分到了不同的应用，这时不允许使用事务，即便允许也无法使用事务，那么应该怎么处理这个问题呢？

首先，服务没有了数据库事务的限制，就不需要考虑数据库事务是放在服务代码还是存储代码了，这是经常发生争论的地方。数据库事务只应该存在于和数据库打交道的存储代码中。

其次，用户在自己的一个操作中连续调用两个服务时，就再也不能假设被调用的两个服务调用是在一个事务中。要保证事务，必须要服务调用方自行来保证，而不是被调用的服务端。既然是要连续调用服务，那么服务的调用方就要清楚以下几种调用的结果：

（1）第一个服务调用不成功，说明调用方的整个调用操作是失败的，不会有问题。

（2）两个服务调用都成功，说明调用方的整个操作是成功的，也不会有问题。

（3）第一个服务调用成功，第二个服务调用不成功。如果业务允许这个情况发生，并且大部分时候是这个情况的话，调用方收到错误提示后自行重试第二个服务调用即可。如果业务不允许这个情况发生，这就是所有人都想用事务来规避

的情况，如转账等这类业务。用户作为调用方需要把第一个成功的服务调用做反向操作，恢复现场。做到这一点需要操作日志的帮助。如果不能恢复现场，必须要人工介入解决，恢复数据。另一方面，如果应用是正常的，失败的原因往往是由于网络或通信的因素而失败，并且往往第一个就会失败，正好处在连续的两个服务调用之间出网络问题的情况非常少。即便发生了，也不需要太多的人工处理工作量。如果应用是有问题的，比如被调用的第二个服务所在的应用崩溃或所在计算机的压力超过容量导致堵塞，也只能人工处理，软件自身是解决不了这个问题的。

希望通过本书的探讨，可以给架构师和软件工程师打开一点思路，引起大家的思考。跳出数据库的思维是可以的，不依赖于数据库的事务也是可以的，未来会更加光明。

道为术之灵，术为道之体；以道统术，以术得道。道与术本为一体：伴随事物发展，道是其本质规律，术则为具体路径；轻道重术或轻术重道皆不可取。但剥开其内在关系，从架构层面重新解读，可诠释为：如能参透架构之本质，洞明技术之本体，就不会再拘泥于旧有的经验实践和笼统的理论条框，能以最直接的方式解决问题，迈入“知其深意，无招胜有招”的境界。《聊聊架构》希望揭开事物的外在“表皮”，再现架构深层之理，向读者揭示最本质的架构之道。

InfoQ精品图书

聊聊架构

ARCHITECTURE

策划编辑：张春雨
责任编辑：徐津平
封面设计：梁金双